University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master Thesis

Utilization of a Graph Database for the Optimization of Wide Cloud Application

Pilsen 2014 Petr Kopač

Declaration

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, May 13, 2014

Petr Kopač

Abstract

This master thesis deals with the comparison of graph and Structured Query Language (SQL) databases and the problems, on which they are optimized, in the context of the cloud application Samepage by Kerio. The software implementation mimics the relevant part, ie. data structures and algorithms related to newsfeeds — both with graph and SQL — with enough precision for the study. The technologies used are above all Java EE, Spring Framework and the databases MySQL and Neo4j.

The testing data is generated for benchmarks, so that one can study traversing of it. Most importantly, searching the access rights in a hierarchical structure for generating the newsfeed. Different orders of magnitude and variability are tried to estimate the computational complexity of implementations. The results are compared in order to find out, which approach is more beneficial for the given problem.

Předmětem této diplomové práce je porovnání grafové a SQL databáze a problémů, na které jsou optimalizovány, v rámci cloudové aplikace Samepage od společnosti Kerio. Softwarová implementace napodobuje relevantní část, tedy datové struktury a algoritmy vztahující se k funkci newsfeed (proud novinek) – jak grafovou tak SQL databází – s přesností vystačující pro tuto studii. Použité byly především technologie Java EE, Spring Framework a databáze MySQL a Neo4j.

Pro účely porovnání se generují testovací data tak, aby bylo možné studovat prohledávání hierarchických struktur s přístupovými právy při vytváření newsfeedu. K odhadu výpočetní složitosti implementací se zkoušejí různé řády velikosti dat a variabilita. Výsledky jsou porovnány k vytvoření závěru, který přístup je vhodnější pro tento daný problém.

Acknowledgements

I would like to thank my parents for supporting me, Tomáš Soukup at Kerio for giving me a chance to work on an interesting and useful project, the colleagues for helping with the consultations and Michael Hunger for the feedback on *Neo4j* optimization. Last but not least I must thank my girlfriend Eliška for giving me motivation relentlessly.

Contents

Part I

Theory

1 Context

1.1 Samepage.io

This thesis studies a real-life problem taken from the application *Samepage.io* by *Kerio Technologies*, which is an integrated site for team cooperation. It is a Java web application that uses standard web technologies and Java Enterprise Edition (Java EE). The web application is very complex, allowing users to collaborate on documents in teams, to follow certain topics, to notify colleagues etc. It is under continual development and recently it has been opened for the public for free with limited capabilities as well as for paying customers.

The application itself or any code from it is not part of the thesis, as it is mostly close-sourced. Therefore, it was needed to program a simulated application data layer with relevant parts to do similar functions. To be able to do so, I was consulting the features personally with my colleagues at Kerio and studying the original code. Of course, some simplifications were made, because of the complexity of the original cloud application. Significant simplifications will be noted in the text, however. Generally, the implementation tries to be as close to the original as possible to the extent of usability of the results for Samepage.io newsfeed.

Simulated data had to be made to fill the databases, for two reasons: firstly, the original customer data would in no case be available due to privacy issues and secondly, the database schemas would be different due to the aforementioned simplifications. But for the benchmarks to be valid, the data has to be coincidental for both databases. The design of the module with generator is therefore generating only once and saving to both database types at the same time. This way we can simulate very easily running the application in two configurations sequentially with enough precision and without the need to re-generate the data.

The real application can be found on the web address: http://samepage.io/

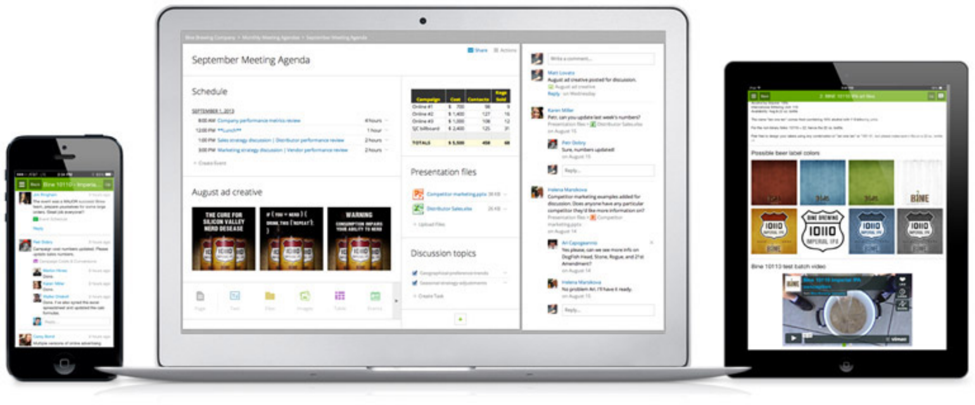


Figure 1.1: This image represents the cloud capabilities of Samepage.io, meaning you can access your data and cooperate with team mates from different devices. On the laptop screen you can see comment stream on the right and content items on the left. Image courtesy by Kerio Technologies.

1.2 The Problem

In Samepage.io the documents are served to the user in a hierarchical tree structure. Not only text documents are treated in this way, but also other data formats, which Samepage integrates — images, videos, calendars, archives etc. We will be using only text documents, as the content itself is not relevant to the problem studied. The nodes of the hierarchy will be called items instead of a document to use the same terminology as in the original application.

The tree structure is very important, however. The access rights for the items are resolved through this structure. When an item has some access rigths settings, all the child items in the tree, are also inheriting those settings, unless they have overriding settings. Even in the original application, the rights are not composite, meaning that to decide what are the rights on the current item, you need to find only the first parent in the hierarchy with some settings.

Another level of abstraction is brought into the picture along with groups of users. The access rights do not have to be assigned only per singular users, but also per groups. But these groups are also hierarchical: let us have a group "Parent" and a group "Child", the latter being a subgroup of the previous, and a user, who is member of group "Child".

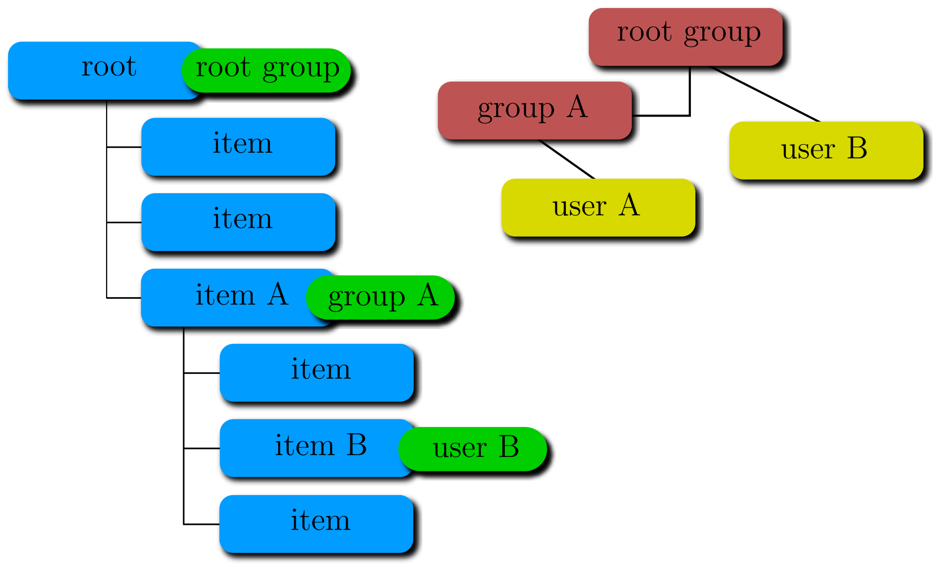


Figure 1.2: The items (🟦) are saved in a hierarchy and each item can have explicitly declared access rights (🟩). If it has no declaration, then the access rights of its ancestor are used. Also the inheritance of rights works in the groups (🟥), so for example *user A* (🟨) can access everything except *item B*; on the other hand *user B* can not access anything under *item A* except for *item B*, where only they have the access right.

The user should be able to access all the items which have reading access for his group, but also for the "Parent" group. In other words, inheritance of rights is at play. See the figure 1.2. Obviously, this is a very simplified example. In reality access rights are set on several users and groups per node and also the relationships can be much more complex.

Now, in this complex environment of hierarchic groups and items with different access right, the user is also following only certain pages, because otherwise it would be too hard for them to keep track of all the information streamed through the whole team workflow.

1.2.1 Newsfeed

”A service by which news is provided on a regular or continuous basis for onward distribution or broadcasting.”
—[Stevenson(2013)]

The users want to be informed about the new comments which other users write on items. Users are asked to comment, if they update significantly a document. Therefore the newsfeed updates the user on important changes as well as recent discussions. There are two possibilities — it can be either the global newsfeed with the most recent new comments from the whole organization, or only from the pages that the user follows. In both cases, users must have access rights to the items to which the comments are attached.

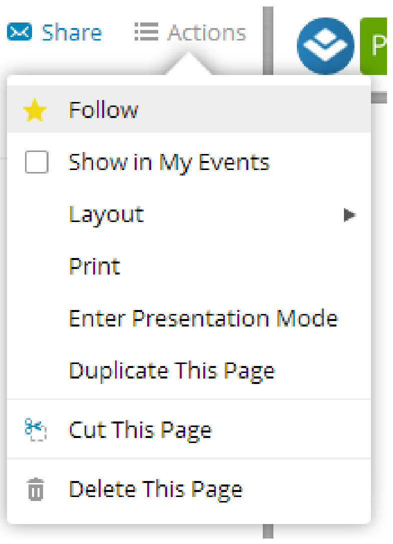
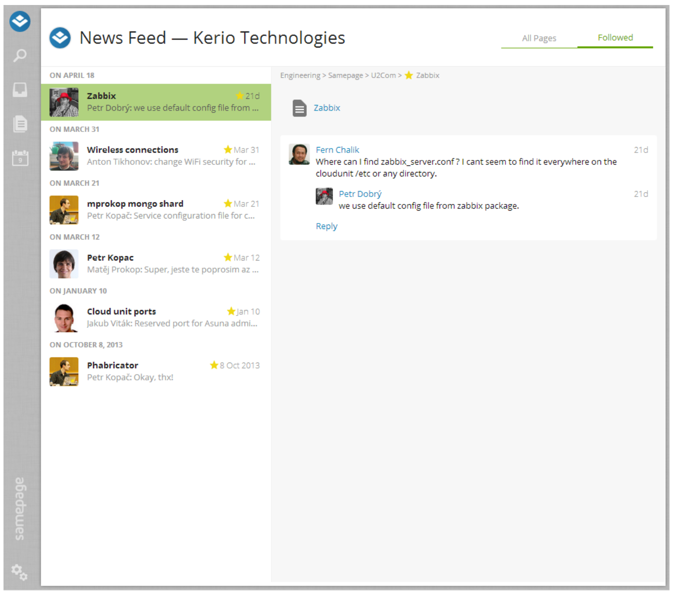


Figure 1.3: Users can follow certain items to keep track of updates.

There are quite random relations, involving especially the follow feature, which is unique per user and therefore can not be cached for an organization as a whole. But not even the global newsfeed can be fetched once and cached. Firstly, this is a live environment: the access rights on individual items are changing and there are new comments every moment. Secondly, each user has different access rights, resulting in different sets of items, where the changes can be monitored.

Finally, we reached the core of the problem — there are several hierarchical data structures that need to be tree-traversed. We are trying to simulate and perhaps find a better solution for it in this thesis, because the approach currently in place takes a lot of computational resources, possibly too many for larger datasets. It is an algorithm development problem, which requests either using extra tables in an SQL database (and then risking data inconsistency) or programming procedures, for example in Procedural Language/Structured Query Language (PL/SQL), because pure SQL does not have the capability to traverse complex structures recursively. What we are considering and comparing to these possibilities is the usage of a graph database.


News Feed — Kerio Technologies

All Pages Followed

ON APRIL 18

Zabbix 21d
Petr Dobrý: we use default config file from ...

ON MARCH 31

Wireless connections Mar 31
Anton Tikhonov: change WiFi security for ...

ON MARCH 21

mprokop mongo shard Mar 21
Petr Kopač: Service configuration file for c...

ON MARCH 12

Petr Kopac Mar 12
Matěj Prokop: Super, jeste te poprosim az ...

ON JANUARY 10

Cloud unit ports Jan 10
Jakub Viták: Reserved port for Asuna admi...

ON OCTOBER 8, 2013

Phabricator 8 Oct 2013
Petr Kopač: Okay, thx!

Engineering > Samepage > U2Com > Zabbix

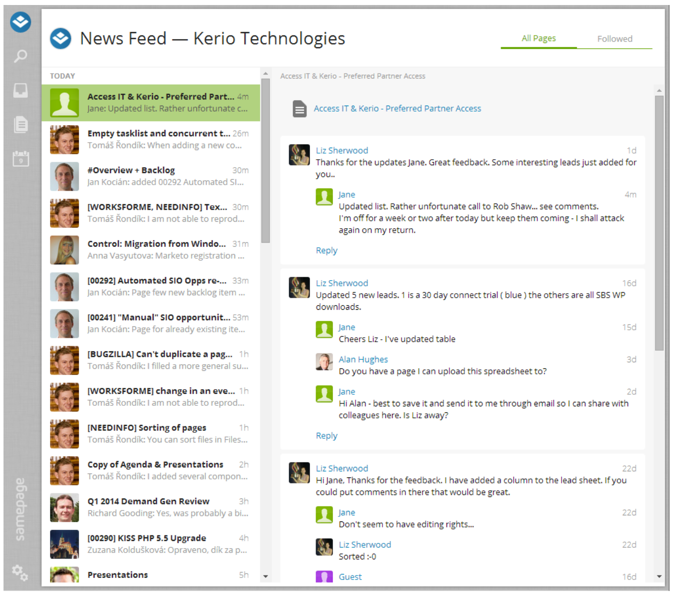
Zabbix

Fern Chalik 21d
Where can I find zabbix_server.conf ? I cant seem to find it everywhere on the cloudunit /etc or any directory.

Petr Dobrý 21d
we use default config file from zabbix package.

Reply

Figure 1.4: In these screenshots, you can see the newsfeed feature in reality. This one is showing only the most recent updates on pages that the user is following. It is uncluttered by noise generated by larger teams.


News Feed — Kerio Technologies

All Pages Followed

TODAY

Access IT & Kerio - Preferred Part... 4m
Jane: Updated list. Rather unfortunate c...

Empty tasklist and concurrent t... 26m
Tomáš Řondík: When adding a new co...

#Overview + Backlog 30m
Jan Kocián: added 00292 Automated SI...

[WORKSFORME, NEEDINFO] Tex... 30m
Tomáš Řondík: I am not able to reprod...

Control: Migration from Windo... 31m
Anna Vasyutova: Marketo registration ...

[00292] Automated SIO Opps re-... 33m
Jan Kocián: Page few new backlog item ...

[00241] "Manual" SIO opportunit... 53m
Jan Kocián: Page for already existing ite...

[BUGZILLA] Can't duplicate a pag... 1h
Tomáš Řondík: I filled a more general su...

[WORKSFORME] change in an eve... 1h
Tomáš Řondík: I am not able to reprod...

[NEEDINFO] Sorting of pages 1h
Tomáš Řondík: You can sort files in Files...

Copy of Agenda & Presentations 2h
Tomáš Řondík: I added several compon...

Q1 2014 Demand Gen Review 3h
Richard Gooding: Yes, was probably a bi...

[00290] KISS PHP 5.5 Upgrade 4h
Zuzana Kolduškova: Opraveno, dík za p...

Presentations 5h

Access IT & Kerio - Preferred Partner Access

Access IT & Kerio - Preferred Partner Access

Liz Sherwood 1d
Thanks for the updates Jane. Great feedback. Some interesting leads just added for you..

Jane 4m
Updated list. Rather unfortunate call to Rob Shaw... see comments.
I'm off for a week or two after today but keep them coming - I shall attack again on my return.

Reply

Liz Sherwood 16d
Updated 5 new leads. 1 is a 30 day connect trial (blue) the others are all SBS WP downloads.

Jane 15d
Cheers Liz - I've updated table

Alan Hughes 3d
Do you have a page I can upload this spreadsheet to?

Jane 2d
Hi Alan - best to save it and send it to me through email so I can share with colleagues here. Is Liz away?

Reply

Liz Sherwood 22d
Hi Jane, Thanks for the feedback. I have added a column to the lead sheet. If you could put comments in there that would be great.

Jane 22d
Don't seem to have editing rights...

Liz Sherwood 22d
Sorted :-0

Guest 16d

Figure 1.5: On the left you can see the list of items and on the right its most recent updates. The global newsfeed is much more alive, with updates in the order of minutes or even seconds, as the organization is hundreds of users large.

1.2.2 Complexity Analysis

Let us try to estimate the computational complexity of the algorithm solving the problem for the worst case scenario. In this scenario, there is only one access right setting on the root node, so the whole path from each node has to be checked. To estimate the time complexity we have to find the average depth of the tree, but that is difficult, as it is a random, not full, multiple-child tree.

We can start with the theory of binary trees: if a tree is completely full, then the number of nodes n is defined $n = 2^k - 1$, where k is then the height of such a tree and 2 is the number of children each node has. According to proof in [Rolfe(2011)] the average depth in the tree is $k_a = k - 1$. This is also true for n-child trees, because the result of the limit (used in the proof) is independent of the number of children. But this is only the lower boundary of complexity, if we know the exact number of leaves.

In reality the average depth of the tree varies. The only fact that has been gained through study of statistical data (and is a kind of heuristic knowledge) is that the trees tend to be much more wide than deep (users tend to create lists, not binary trees). Therefore it should be a good choice if we select $k_a = log(n)$ as a credible boundary for average usage.

In the case of the global newsfeed, we have to check n nodes, while in the case of followed items only the transitive closure of their children f^+; we never know, however, how large the subtrees f^+ are. Generally, the size of followed subtrees should be much lower than all the nodes: $f^+ << n$. Still, it adds the necessity to find all the children, which is $O(f^+)$.

Now, that we know the range and average depth, we can estimate the upper boundary of checking access rights. If we were not in the worst case scenario, where there are no special access rights, there would be only some nodes (plus the root), where the user has access, that is reflected with n_x:

$$Check_{access} = O(n \cdot k_{ax} \cdot n_x)$$

Where k_{ax} is the average depth of an item under a directly accessible node and n_x is the number of directly accessible nodes. Although this seems as adding complexity, these two number are usually very low, unless there would be very complicated access right structures. In fact, the most uncomplicated access rights result in the worst case scenario, where the average depth is the one of the tree and there is only one directly accessible item, the root.

$$Check_{access} = O(n \cdot k_a)$$

At this point, the algorithm has determined, which items are relevant and at the same time accessible through access right policies. But we need to fetch only a few newest comments on the items. Let us estimate roughly that the total number of comments in the hierarchy reaches around the same order of magnitude as the number of items (or is linearly dependent) and the same is valid for any subtree (in average). Then $comments \approx n$. Finally, we need to order them all, which in worst case for Quicksort is $O(n^2)$ (Quicksort is used in MySQL[Oracle(2014)]).

This gives us the complexity of:

$$O_{followed} = O(f^+ \cdot (1 + log(n) + f^+))$$
$$O_{global} = O(n \cdot (1 + log(n) + n))$$

If we consider that f^+ is a linear function of n and also, $log(x) << x^2$, or more formally:

$$\lim_{x=0 \to \infty} \frac{log(x)}{x} = 0$$

We can simplify the big O notation into the conclusion:

$$O_{followed} \approx O(f^{+^2})$$

$$O_{global} \approx O(n^2)$$

1.2.3 Expected Enhancements

As for the graph database, we expect it to be able to use alternative, recursive queries and make use of the graph structure, which could fit the described environment better than the SQL table structures.

If the database is capable of proper optimizations and we are able to use them, the migration to graphs could speed up searching the trees and generating the newsfeed. At this point we are not aware what the possibilities are exactly, so it is needed to investigate them and employ adequately.

We are not looking for an absolute speed gain, however. The important part of the results is the *behavior with growing order of magnitude.*

We are mainly interested in the statistics — whether with the growing number of items, users and groups the graph database can handle the complexity better, the same or worse.

The extraction of updates is currently done with complex SQL queries that rely heavily on application-side filtering and handling. It is possible that a graph database could allow us to do all the necessary operations directly, utilizing its built-in optimizations and lowering network communication as it would transfer back to the application only the useful dataset.

But the most important and interesting aspect is that we are given an opportunity to try lowering the complexity — if the database could cache the nodes and relationships in such a manner that matching the paths between them would become less than exponential, or at least allow to run efficiently enough on millions of items.

2 Databases

The database forms the storage layer of the application. In this case, we can not even use in memory storage, because the magnitude of the data present will be exceeding the capacity of today's normal computer (4 GB). But fortunately, that is no problem as it brings us even closer to the original application, even though disk operations will slow the process of simulation down.

A simplification has been made here, however, because the original software uses, of course, a distributed storage, whereas the model uses only a single instance of the database, whether it is SQL or graph. But this should not be an issue, because we are focusing here on fundamental differences in algorithm development and complexity between SQL and graph (noSQL) databases.

2.1 What is an SQL Database?

The history of SQL goes back to 1970s, when the predecessor Structured English Query Language (SEQUEL) was developed at IBM.[Chamberlin(1974)] As a standard, SQL was adopted in 1986 and since then several latter versions were created. Currently, there are more than just one canonical version of SQL in use. They differ usually in extensions, which are common in commercial high-end Relational Database Management Software (RDBMS), adding enhancements to their databases as for example the PL/SQL language by Oracle, which adds procedural programming capabilities, allowing execution of user-defined programs in the database engine context.

The SQL is a declarative language. In short, you can declare what should be retrieved, or inserted etc., but you are not directly writing the algorithm, although you may significantly influence the optimality of the query by using different approaches, indexing, alternative commands or even the right ordering or embedding. Nevertheless, SQL gained popularity and became the traditional approach to save large amounts of data in computer science. The main feature of SQL are the table structures accessible by queries and related by foreign keys; therefore this is called the relational model. Also, it is notable that all the members of a given table have the same structure dictated by the table, which is in most cases rigid and rarely changed as a part of ordinary operation of applications using the database.

For the sake of this simulation the MySQL RDBMS will be used. One reason for that is this database is at the moment one of the two most used RDBMS overall; second, but maybe even more important is the fact that it is used in the original application. We will try to benefit from its capabilities as much as possible, not considering compatibility with other database systems necessarily.

2.1.1 Recursive SQL

Basics of SQL will not be described here.[1] Instead, I would like to present the reader with the results of my research on the current possibilities of recursive traversal of data structures in SQL databases, as it is the core of the problem. Above all, the capabilities of MySQL will be examined for practical reasons, but also compared with other popular databases.

It is worth mentioning that any recursive algorithm is translatable into cycles, but that is again procedural, not declarative programming. Also, in the scope of different programming languages, which have different capabilities of expression, it can be difficult or even impossible due to restrictions of the language or the engine.

Stored Procedures

MySQL contains support for procedures and functions. These are very restricted, however. For example you can not `SELECT` a result set from the database and use it programmatically for the next iteration, unless it is a simple value as a string or a number; arrays are not supported. It is not even possible to make `UNION` to merge these results, so if one calls subsequently several `SELECT` commands, the client will receive several result sets, which have to be used directly with the Java Database Connectivity (JDBC) API, because higher abstractions are usually built only for one result set.

Although it is possible to call procedures recursively, it is disabled by default and must be allowed by setting maximum recursive depth to a number higher than 0. But again no support of arrays and using the data sets from `SELECT` restricts the possible uses to minimum.

[1]For learning the basics using some on-line interactive course can be recommended; there are plenty of them, for example http://www.w3schools.com/sql/

```
SET @@GLOBAL.max_sp_recursion_depth = 255;
```

Recursive SQL Construct `WITH RECURSIVE`

This construct has appeared in the SQL standard already in 1999[2]. It uses Common Table Expression (CTE), which is a temporary result set in scope of the executing SQL command. Another synonym is "subquery factoring". The list of databases with this feature can be found on-line (for example at [CTE(2014)]), but to name just a few: *PostgreSQL* (since 8.4), *Microsoft SQL Server* and *Oracle* (since 11g release 2). Unfortunately, *MySQL* still (version 5.7) does not support this feature.

Below you can see an example taken from Wikipedia, as I could not use this in the selected database. In the example the `(n, fact)` are input parameters and we can see the recursive calling with the `FROM temp` expression. This results in a 2-column and 9-row result set with factorials.

```
WITH RECURSIVE temp (n, fact) AS
  (SELECT 0, 1 -- Initial Subquery
    UNION ALL -- Merging into one result set
   SELECT n+1, (n+1)*fact -- Recursive return values
    FROM temp -- Recursive Subquery
    WHERE n < 9) -- The ending condition
SELECT * FROM temp; -- Return results
```

There actually exists an emulated `WITH RECURSIVE` which I found while researching on the Internet (in [Bichot(2013)]), but although it does solve the problems that were mentioned above, it is quite not a clear solution, which would be suitable for production systems. Despite that I tried to run it (from pure curiosity), but it ended with error, probably because of some incompatibility with my MySQL version (5.6) or configuration, as it uses dynamic SQL heavily (creating commands on the fly from text strings).

[2]ISO/IEC 9075-*:1999

Repeated Calls

If we can not use any of the two possibilities mentioned above, the only approach that we can resort to is having the recursive code on client side, sending repeatedly individual requests to the database system. This approach transfers more data over the network between the application and the database, having possibly bad influence on performance as more and more queries are sent, but on the other side the programmer receives much more control over the algorithm design.

In the project the first and the last approaches were combined, because using the `WITH RECURSIVE` was not possible. See the Implementation part, chapter 5 for more details on how SQL was used to solve the problem.

2.2 What is noSQL?

Although the SQL language and related RDBMS have become essential for data storage, in recent years a movement towards other possibilities has become very influential. That movement is called *noSQL* and comprises all the different approaches to make data persistent without using SQL. The shift in paradigm was caused by many factors — a need to store big data, a desire to have a simplified database, or to capitalize on the graph characteristics of social networking and semantic data.

The only noSQL described further besides this section will be *Neo4j*, which uses graphs, is the most used in its category and is used as an alternative to SQL in the model application. To make a short overview of the current noSQL possibilities, I added this short list, which names always just one example to get the general knowledge. There are 4 fundamental species in the noSQL universe: *column, document, key-value* and *graph.* Let us see a few quick examples:

MongoDB (document) Saves data in JavaScript Object Notation (JSON) format, or more precisely in Binary JSON (BSON), which are fundamentally associative arrays, or documents, as called in this database's jargon. As the name already hints, the language predominantly used to access the data is JavaScript. Also, it does not use tables with rigid structure; instead, data is contained within collections, and the units of data itself are called documents instead of rows to highlight the complexity.

The documents have no prescribed structure whatsoever and members of one and the same collection can be of completely different nature. An interesting advanced aspect is the possibility to run JavaScript code in certain cases on the database server. Let us see a few sample commands:

- `db.getCollectionNames()`
  gets a list of all the collections present in the current database schema,
- `db.people.find({"age" : {"$gt" : 18}})`
  finds and returns all documents from collection `people` with the property `age` higher than 18.

Accumulo (column) The column databases are distinct from the relational model, having columns that are not always present in all rows of a table, which itself is a very vague concept in their context. They are usually useful for saving and retrieving large quantities of sparse data. Examples of *accumulo* shell and Java Application Programming Interface (API):

- `createtable test`
  (in shell) a table `test` without any columns is created
- `(new Mutation(rowID))`
  `.put(colFam, colQual, colVis, timestamp, value);`
  (in Java) creates a mutation, which manipulates a row of a given ID (simlar to `UPDATE` in SQL).

Redis (key-value) Actually an associative array of keys and values, the key-value storage is used to solve the dictionary problem. Redis is currently the most popular key-value storage and stores data in memory. It is possible to run scripts written in Lua language on it. A shell is available and also other different APIs.

- `set mykey "my value"`
  (in shell) a key `mykey` is created and has content of type string `"my value"`
- `redisClient.set("mykey", "my value", callbak)`
  (NodeJS API) this creates the key-value pair as the previous example and as it finishes the callback function is called.

Neo4j (graph) The graph database is even more different from the previous noSQL types, because it is built on the principle of nodes connected with relationships. There is no fixed structure of them and therefore it is very flexible.

Nevertheless, the nodes and relations can have labels (something like classes). It uses its own query language named *Cypher* and here we have again two examples:

- `MATCH (n) RETURN count(*)`
  matches nodes, without constraining to any properties. That makes the query select all the nodes and return their count using the built-in function.
- `CREATE (me:User {name: "Bob"})`
  creates a node labeled as a `User` with the property `name`.

This was just a short list to get the feeling of alternatives to the standard SQL solution; further, we will discuss only noSQL concepts relevant to the graph database *Neo4j*, which was selected for this project.

2.3 Neo4j

Neo4j has been mentioned in the above section as one of the representatives of the graph databases. It was chosen as the platform to try optimizing the data layer for several reasons:

- it is the most popular graph database,
- it has its own advanced, but clear query language *Cypher*,
- it is very well documented, including a free on-line video tutorial with web application allowing the student to directly try out *Cypher* queries,
- there is a Spring Data project, which allows combination of Spring technologies with Neo4j.

Unfortunately, the Neo4j server does not allow any binary protocol as SQL databases do. They do have their language-specific drivers and the abstraction of JDBC and Open Database Connectivity (ODBC). For remote access Neo4j offers two possibilities:

- REpresentational State Transfer (REST) API over Hypertext Transfer Protocol (HTTP) - which is unfortunately slow and allows using transactions since only recently,[3]
- writing own API based on an embedded Neo4j server.

As mentioned in the second option the server can be also embedded in an application. That means the server instance is running directly within the memory space of the application, and the program can use its direct Java API. The data is not stored in memory, but on a hard drive; the memory is used only for caching the data and operations above the data; therefore the embedded version needs access to a persistent storage.

For the project I chose the embedded version, because using REST API proved to be cumbersome; on the other hand, writing my own API would take too much time. All in all, it is a viable variant.[4] But we must also note that using an application-embedded database approach gives Neo4j a potential advantage (having shorter method calls than communication on the network or socket), that is why we should not compare the absolute amounts of time taken to process operations directly, but rather consider the development of complexity, when we send higher quantities and more structured data.

2.3.1 Cypher Language Quick Overview

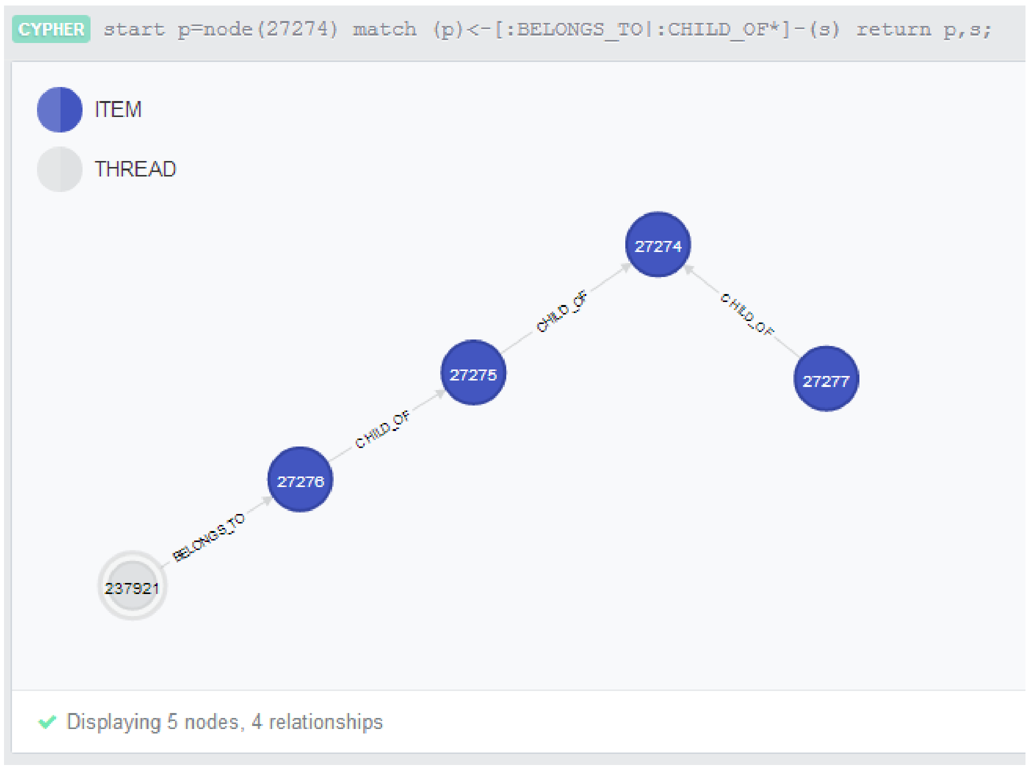
Similarly to the SQL language, *Cypher* is the language used to fetch or change the data for Neo4j. Most of the commands have their equivalents in these two languages, but what is a new and intriguing part is the graph matching expression. We can see that the language itself gives the programmer an efficient way to express relations.

The following command will match any nodes with a relationship between each other and return them. Not only one pair, but all of the pairs:

```
MATCH (n)-->(m) RETURN m,n;
```

[3]In version v2.1.0-M02, but for the project it was decided that using unstable variant is not a good idea, so the stable version 2.0.2 was used.
See: http://docs.neo4j.org/chunked/milestone/rest-api-transactional.html

[4]It was also needed to restart the database programatically due to fast non-transactional bulk insertion of nodes, so that was one more reason for the embedded version.



```
CYPHER start p=node(27274) match (p)<-[:BELONGS_TO|:CHILD_OF*]-(s) return p,s;
```

Figure 2.1: Example nodes — you can see the blue nodes, which are labeled `ITEM` and a gray one labeled `THREAD`. These nodes can be related to each other by relationships, which also can be labeled (`CHILD_OF` and `BELONGS_TO`); both nodes and relationships have unique IDs and can have properties.

Will match nodes labeled as `ITEM`s[5] with a relationship of type `CHILD_OF` and return only those that have a property `name` with value `"Test"`:

```
MATCH (c:ITEM)-[:CHILD_OF]->(p:ITEM {name: "Test"}) RETURN c,p;
```

First, find the node with a given ID, then find all of its children recursively; the asterisk marks that there can be an infinite repetition of this relationship, an infinitely long path. That is useful, but can be also *dangerous* if the data contain cycles or is very large:

```
START n=node(59877) MATCH (n)<-[:CHILD_OF*]-(p:ITEM) RETURN n,p;
```

Usually, we do not want thousands of results, but only the first or last few; that is done the same way as in SQL, for example here we order the result in descending order and return 21st to 30th item:

```
// ...
RETURN c ORDER BY c.updated DESC SKIP 20 LIMIT 10;
```

We can also combine matching with creating; one can for example use these patterns to create relationships between nodes. And if we commented out the `WHERE` clause, it would create that relationship between all the `USER` nodes.

```
MATCH (a:USER),(b:USER)
  WHERE a.name = 'Peter' AND b.name = 'Bob'
  CREATE (a)-[r:KNOWS]->(b);
```

2.3.2 Typical Problems Solved with a Graph Database

Transitive Closure

One of the most interesting parts of the graph approach is the capability to do recursive traversal. The transitive closure can be explained as follows: if we

[5] As you can see, the notation goes like this: `(variableName:labelName)`

have a list of towns and a list of road between them, the transitive closure of this relation is a list of pairs (towns), which it is possible to reach from one another with arbitrary distance on the road.

If we think about how we are dealing with relations in the traditional relational model (SQL), we might notice that we only work above rigid length-specified relations. But our problem described in 1.2 requires a completely different approach — recursive climbing on a tree structure with undefined depth. This can be expressed in *Cypher* very naturally with just one pattern matching, but not so easily in SQL (see section 2.1.1).

Semantic Web and Social Networks

Today's big challenge of the web is the *meaning* of the data. Increasingly, we are trying to work over larger and larger amounts of interconnected data. This data is characterized by non-fixed structure and vast relationships. Although the semantic web itself is only slowly emerging, one phenomena has already risen to vast success — the social networks. These networks actually contain unprecedented amounts of semantic information about its users, their relationships, preferences etc. Even already from the name itself we can imagine relationships are the core principle here. The graph databases fit this with their architecture of interconnected nodes.

Traditional Graph Algorithms

Some of these algorithms are already implemented in the graph databases and we can expect that there will be even more of them in future; instead of writing them over and over again, we would use the already present ones operating directly over the data with very simple queries. For example, to find the shortest path in Neo4j, we would use `MATCH p = shortestPath((a)-[*..15]-(b))` (finds shortest path of maximum length of 15). To name just a few:

- Shortest path problem (Floyd-Warshall, Dijkstra's algorithms)
- Breadth-First Search (BFS) and Depth-First Search (DFS)
- Graph kernel

Part II

Implementation

Implementation

The implementation is heavily influenced by the original application environment and libraries. That is why Java EE was chosen as the main platform, Tomcat as an application server and MySQL as SQL RDBMS. But on the other hand, I also wanted to re-use my knowledge from programming web applications with Spring framework and especially explore the framework's potential with the Neo4j graph database.

Thanks to the support of the libraries it was very easy to use several important design patterns, which make the application much more readable and the code reusable, for example the Model-View-Controller (MVC), described in the section 3.3 on page 27, a basic web application.

In this part, multiple concepts from the libraries used in the project will be described for the reader to better understand the abstractions used in the implementation.

Further, we will walk through creation and optimization of the used SQL and *Cypher* queries to better understand, how they work.

And finally, the object model of the application and individual parts will be described and explained in detail.

The code examples will be written in a simplified form, even though they are taken from the implementation. In other words, they should go straight to the point. This is for reader's comfort and trying to lower distractions by non-essential code.

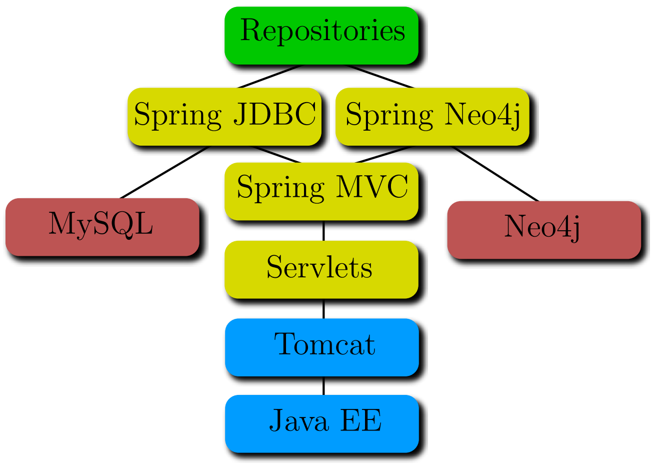


Figure 2.2: The application is built on several layers of library support. On the lowest level there is only *Java*, but then on top of it runs *Apache Tomcat* (servlet container), which makes it a web application. *Spring MVC* adds the mentioned layering of application and *Spring JDBC* together with *Spring Neo4j* provide abstraction levels on top of connections to respective databases. The repositories, which are on top of this hierarchy and are responsible for manipulating the data, can then make full use of the provided abstractions.

3 Spring Framework

The Spring Framework is a vast platform of libraries for Java EE. There are many modules, but I used mostly the part around web applications (*Spring Web MVC*) and Neo4j (*Spring Data Neo4j*). Besides that, the basic principles of this framework were utilized, above all the dependency injection mechanism, together with bean declarations and Java configuration. In the following three sections the principles will be described, but for more material on the topic the reader can consult multiple books written on the topic, for example [Walls(2011)], or the official documentation, which is accessible online at: `http://docs.spring.io/spring/docs/current`

3.1 Dependency Injection

The dependency injection is a very useful mechanism (and the core of Spring). Equivalent design pattern is named Inversion of control (IoC):

> A software architecture with this design inverts control as compared to traditional procedural programming: in the traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the reusable code that calls into the custom, or problem-specific, code.
> —[IoC(2014)]

Spring is an IoC container. It allows the programmer to configure the application context, declare a bean and then use it in any other software component of the same context.

The software components must oblige another design pattern, which is called a bean. It is basically a class with a standardized interface — get/set methods and standard constructor (no arguments). It is important that the name of fields correspond with the methods, eg. field `myNumber` should have `getMyNumber` and `setMyNumber` methods. This is actually very easy to prepare, because most modern Integrated Development Environment (IDE) provide automated generators.

Later, if one software component needs another one, the Spring application context can ”autowire” the dependency on runtime, if the programmer declares it properly. There are basically two ways - you can either use Extensible Markup Language (XML) configuration or Java configuration. In the implementation, only the Java configuration is used, except of course for the database connection settings, which are externalized.

The programmer can also easily declare such dependencies directly in the code using annotations `@Autowired`. The dependency is then searched using the type (or also the name of the field, if more instances of such type exist in the context) and set on runtime — with this approach we do not even have to create get/set methods. Also, the `@Autowired` annotation can be used on methods or constructors. The general principle of annotations in Java is a form of declarative programming.

Since the previous version of Spring it is not necessary to declare the software components in separate XML files anymore, as only two simple things replace this mechanism. Firstly, you need to declare that you want use the component-scan capability of Spring, with a base package for recursive scan - you can do it in application context XML or in Java configuration with `@ComponentScan(...)`. Then you annotate all your component classes with `@Component`, or its children, for example `@Repository`, which is very useful for Data Access Object (DAO) classes, which manipulate the data and with this mechanism, their instance is serving anywhere you need it.

3.2 Java Configuration

> The central artifacts in Spring's new Java-configuration support are `@Configuration`-annotated classes and `@Bean`-annotated methods. The `@Bean` annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container.
> —[Johnson(2014)]

In the code example you can see the configuration used in the implementation, which is a class that enables MVC essence as well as component scanning (except for other configurations, which is what the exclude filter does). There is also a very important annotation `@EnableTransactionManagement`, which declares that from now on any method annotated with `@Transactional` will be running in a transaction.

```
@Configuration
@EnableWebMvc /* The same as <mvc:annotation-driven> */
/* Load only non-configuration classes. */
@ComponentScan(basePackages = {"cz.pkopac.thesis"},
  excludeFilters = {@ComponentScan.Filter(
    type = FilterType.ANNOTATION,
    value = Configuration.class)})
@EnableTransactionManagement
public class MvcConfig extends WebMvcConfigurerAdapter {
  ...   /* Bean declarations would go here. */
}
```

It extends `WebMvcConfigurerAdapter`, because there are already some useful beans declared, so that we do not have to repeat ourselves each time with a new web application. Speaking of which, we can now have a look, how a bean is declared in our configuration class:

```
@Bean
public NamedParameterJdbcTemplate jdbcTemplate(DataSource
    dataSource) {
  return new NamedParameterJdbcTemplate(dataSource);
}
```

The class `NamedParameterJdbcTemplate` is a useful interface for translating a Domain Object (DO) into query parameters for SQL. But before we can use it anywhere, it has to be configured, namely it needs the JDBC data source, which is given here as a method parameter. The instance is then injected by the IoC container, because it automatically searches for such a bean reference, which fulfills the `DataSource` interface and binds it to this method. The data source is declared in another method with database connection configuration. The way to get the reference to this database interface instance:

```
@Component /* Without component annotation, the container wouldn't
              know that it needs to autowire this class. */
public class SomeClass {
  @Autowired
  NamedParameterJdbcTemplate npJdbcTemplate;
  ...
```

The reference can be used as soon as the class is constructed. If the initialization of the class would need it a `@Post-Construct` annotation can be given to a method to do the necessary operations.

Combining with XML configuration

There is a possibility to combine the Java configuration with XML configuration. It depends, which one is the primary, ie. which one loads the other. Both approaches are possible and equivalent; in Spring Framework Reference Manual [Johnson(2014)] one can find detailed description. I chose Java configuration and imported only the settings for the SQL database connection:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<!-- Defining doctype is important, as otherwise the XML is not
    recognized as valid. -->
<beans>
 <bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
  <property name="url"
     value="jdbc:mysql://localhost:3306/pkopac?..."/>
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="username" value="..."/>
  <property name="password" value="..."/>
 </bean>
</beans>
```

Then, the programmer just needs to let the IoC know, where it should load the resource from, which is again done declaratively with an annotation. Notice the prefix "classpath:"[1] — Spring Framework supports several different path prefixes, which allow to utilize different scopes.

```
@ImportResource("classpath:cz/pkopac/thesis/config/datasource.xml"
    )
public class MvcConfig //...
```

[1]Classpath in Java is the place, where the compiled classes are stored; it can be inside a Java ARchive (JAR), on a file system or even in memory.

3.3 MVC

The Spring Framework provides a module with special support for web applications. It was used in the implementation as the container for running the experiments, accessing the data layer and serving the results in visualized form to the user. After all, it is the closest platform to the original cloud application, much closer than eg. running the tests in command line. You can have look on a scenario, which takes place in here on figure 3.1.

> The Spring Web model-view-controller (MVC) framework is designed around a `DispatcherServlet` that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files. The default handler is based on the `@Controller` and `@RequestMapping` annotations, offering a wide range of flexible handling methods. With the introduction of Spring 3.0, the `@Controller` mechanism also allows you to create RESTful Web sites and applications, through the `@PathVariable` annotation and other features.
> —[Johnson(2014)]

3.4 Database Interfaces

3.4.1 Transactions in Databases

In the database terminology a transaction is a unit of requests or queries to the database, which must have a certain level of independence and separation to other requests. Mostly, total separation is used, but there are cases, as for example in distributed or sharded databases, where it is beneficial to loose some independence at the gain of speed.

A common term is used to describe the transaction properties: ACID — *atomic, consistent, isolated* and *durable.* In the thesis transactions are used complete or none, so we will not discuss any other types any further.

A transaction must be explicitly started and explicitly ended.

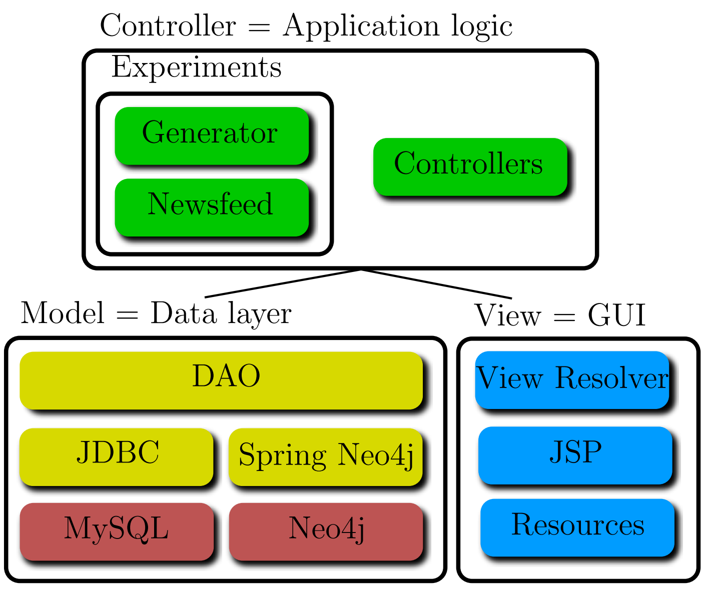


Figure 3.1: As you can see, the experiments are running under a controller as a part of application logic. They interact directly with DAO and create results, which can be visualized. The controller is serving the results (and progress information) to view, which is Java Server Pages (JSP) — that generate Hyper Text Markup Language (HTML) and use Cascading Style Sheets (CSS) and JavaScript (JS) files from resolver of static resources.

```java
public void doSomeOperations(Object[] objects) {
  try (Transaction tx = database.beginTx()) {
    /* Database operations would go here. */
    tx.success();
  }
}
```

In this case, taken from one of the DAO, we can see that the transaction is started, then some operations on the database are done and then, provided that everything works out and no exception is thrown, the code ends with closing the transaction as successful. This is called *committing* which means that from now on all the changes made are being reflected by the database.

As for the the ”roll back” operation, it is done automatically in case of an exception. The construct used in the example is new in Java, therefore it might be worth explaining. It is *try-with-resources*, and if the resource implements an interface which basically just gives a point of entry for an exception, it can be used to spare repeated code. So at the end of the `try` block, if the method `success()` has not been called, all the operations invoked within the block are reverted. In any case at the end all the used resources are released properly, as would one do with a *close()* method.

The transactions are not supported by all databases (some noSQL do not have them), but they are used in MySQL and Neo4j. While programming with transactions, the programmer needs to be also aware of the limits — some databases have the number of transactions limited, the size of all running transactions and the size of a single transaction. If those limits are passed, it usually results in the loss of data, so while working with big data, it is necessary to do commits after certain number of rows.

Also a worthy note is that transactions speed up bulky database operations. With *autocommit* settings, the data is committed after each query. This might be useful for simple straightforward usage, but becomes very problematic, if we want to make any wider changes to the database. For example, let us say that we want to save one thousand entries in an SQL table. What happens with *autocommit* set on is that after each update of the table we also update indexes, check integrity and other costly operations, which could be done only once at the end; lowering effectively the speed by around two orders of magnitude.

3.4.2 Transactional Annotation

For the SQL parts of DAO I chose to use the annotation `Transactional`, which automates transaction handling for the programmer. It not only saves writing of repetitive code (compare the example below and the one from 3.4.1 on page 27), but also allows to write the code only once and set strategies and other settings through a transaction manager in configuration. Also, in reality there are many different transaction infrastructures and this provides a unified interface.

```
@Transactional
public void doSomeOperations(Object[] objects) {
  /* Database operations would go here. */
}
```

When a class, interface or method is annotated, the Spring Framework injects parts of code, which open and close transactions and of course handle errors. This code is injected using *AspectJ*, which is Aspect-Oriented Programming (AOP) extension of Java. *AspectJ* is currently standard for AOP in Java and uses very close syntax, but it is completely handled by Spring and transparent for the programmer.

To make it work for the application, the configuration of Spring context must know, we want this enabled. There is the `@EnableTransactionManagement` annotation, which one needs to put on the configuration class (`MVCConfig` in our case). By supplying parameters to this annotation the transaction manager can be configured.

3.4.3 Manual Transaction Handling

While it is possible to use annotations for transactions, and it is very simple to set up the basic scenario of one database access, it becomes more complicated, when we want to have access to more databases and especially of different type.

Setting up two annotation-based transaction managers (one for SQL and one for graph database), according to the documentation, both should be working right, if the programmer supplies two annotated configuration classes for transaction management. While declaring methods, which operate over one of the databases, one has to specify explicitly in the annotation, which manager to use.

Unfortunately, I did not manage to set this up into functioning state, so I had to fall back to manual handling for Neo4j in my code. But after further development and optimizations, it actually turned out not to be a big problem, as the insertions in the generator part had to be done *without* transactions — through special low level API, which is described in 4.2 on page 44. Transactions for Neo4j are used only in the part that fetches the newsfeed for a user.

3.4.4 Spring JDBC

JDBC is the standard of connecting to SQL databases in Java. The main advantage is the added level of abstraction, which allows you communication with any SQL database work with the same API. The only thing needed is to have the right driver for the database on Java classpath[2], JDBC then loads the class and uses it for translation of the requests into proprietary protocols of the databases.

What Spring JDBC adds is yet another level of abstraction, this time with much wider possibilities for the programmer than just running queries in SQL. To give a practical example, I used the automatic object mapping, while inserting objects into SQL database. You can see how simple it is in the following listing:

```java
@Transactional
public void sqlInsert(Object[] objects, String tableName) {
   SimpleJdbcInsert inserter = new SimpleJdbcInsert(sqlDataSource).
     withTableName(tableName);
   BeanPropertySqlParameterSource data[] = new
     BeanPropertySqlParameterSource[objects.length];

   for (int i = 0; i < objects.length; i++) {
      data[i] = new BeanPropertySqlParameterSource(objects[i]);
   }
   inserter.executeBatch(data);
}
```

[2]The full class name has to be specified as well while creating the connection, because there might be more drivers and more connections, ergo some automatic loading would be tricky.

The class `SimpleJdbcInsert` provides a mechanism, which basically looks up the table structure in the SQL database, creates `INSERT` statement and fills it with data from `SqlParameterSource`, in this case `BeanPropertySqlParameterSource`, which expects a bean instance (see 3.1 on page 23) — from that it extracts the properties (or columns in SQL terminology).

3.4.5 Spring Data Neo4j

This project[3] integrates Neo4j into the Spring Framework. I used it in the beginning of the development, but later found out, I would need to fall back on lower API to be able to do inserting of nodes really fast.

Due to the special nature of the project, Spring Data Neo4j was not actually very useful, as the business logic basically needs to upload millions of nodes and relationships in a short period of time.[4] As it proves, with more abstraction layers one gets also less efficiency. But for standard application development it is probably very useful as it can handle the transactions, domain object transformations etc. To give an example, there is a code snippet that was used in the batch DAO before it was exchanged for `BatchInserter`:

```java
import org.springframework.data.neo4j.support.Neo4jTemplate;

//...

public void neoInsert(Group[] groups) {
  try (Transaction tx = graphDb.beginTx()) {
    for (Group group : groups) {
/* With the following a bean instance is "simply" transformed into
    a node. One can supply annotations on the bean class, which
   can even contain Cypher queries or other hints. */
      Group u = neo4jTemplate.save(group);
    }
    tx.success();
  }
}
```

[3]Official website: http://projects.spring.io/spring-data-neo4j/

[4]Very insightful series of articles on this topic by one of the programmers of the Neo4j team, Michael Hunger: http://jexp.de/blog/2013/05/on-importing-data-in-neo4j-blog-series/

4 Neo4j

In this chapter, we will get a look on the structure of data through the interactive Neo4j web interface (web console), analyze the queries, learn about how `BatchInserter` was used and look into other related topics. But first let us have an overview on our data structure as seen by Neo4j. Firstly, it is important to mention that in Neo4j, not all the nodes, not even those of the same type (label), have to have the same columns. Logically, the equivalent to having a `null` value in SQL is not having the column at all in Neo4j node. But in this case the same structure is kept for all the nodes of a same type, except for the fact that the columns related to relations are not present, as they are actually forming relationships (directed edges of the graph).

Labels

As defined in `cz.pkopac.thesis.neo4j.Labels`:

ITEM is one data node, can be `CHILD_OF` another `ITEM`. Also, has an owner (relationship `IS_OWNED_BY`).

GROUP represents user group with privileges, can have relation `HAS_ACCESS` to an `ITEM` or to another `GROUP`, which effectively means it is its child and inherits all the access rights from it.[1]

USER represents one user. The user can be `MEMBER_OF` a `GROUP`, can have direct access onto an item through `HAS_ACCESS`, can have `FOLLOWS` on an `ITEM` and can be owner of `ITEM`, `THREAD` or `REPLY`.

THREAD is a newly created discussion topic on an `ITEM` — this relationship is declared as `BELONGS_TO`.

REPLY is a comment, which replies on a topic and is `CHILD_OF` a `THREAD`. Most importantly, these two labels have a property `updated`, which is used for sorting in the newsfeed.

[1]Originally, there was also relationship `CHILD_OF` as with `ITEM` structure, but later I realized that having a continuous path of `HAS_ACCESS` can be employed for having a simpler and efficient *Cypher* query. On the contrary, having the same relationship as with *ITEM* would result in hard-to-constrain path resolution.

Relationships

As defined in `cz.pkopac.thesis.neo4j.Relationships`
(only listing possibilities, for details see the previous list):

MEMBER_OF `(:USER)->(:GROUP)`

IS_OWNED_BY `(:ITEM)->(:USER)`, `(:THREAD)->(:USER)`, `(:REPLY)->(:USER)`

HAS_ACCESS `(:USER)->(:ITEM)`

BELONGS_TO `(:THREAD)->(:ITEM)`

CHILD_OF `(:ITEM)->(:ITEM)`

FOLLOWS `(:USER)->(:ITEM)`

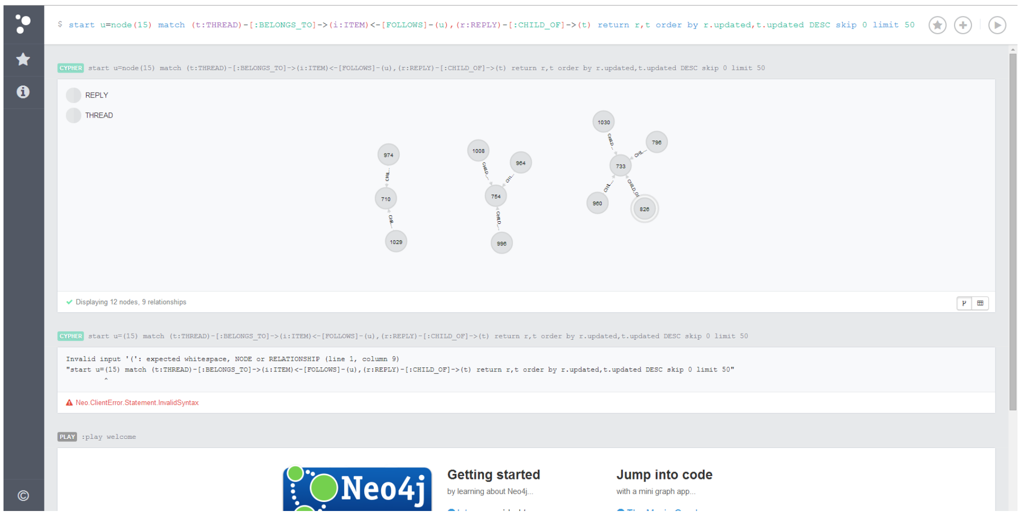
4.1 Queries for the Newsfeed

The Neo4j Community comes with a handy utility, which allows the user (in this case database architect or data layer designer) to interact with the database directly with queries, as *MySQL Workbench* does. Once you have successfully started the standalone version of the database server, it offers you a Uniform Resource Locator (URL), which will lead you to a web interface very useful for directly testing your queries (see fig. 4.1).

Let us go now through several queries and accompanying visualizations, not only to better understand the domain of the problem, but also to learn how the queries used in the application were built up.

In the first picture (fig. 4.2) we can see simple selection of groups, which are hierarchical. It selects (or better said in Neo4j terminology — *matches*) actually just the first ones, they're not even at the top of the group tree — which does not really matter as for the database it is a general graph, not a tree. It is limited to see just a few, not hundreds of them. The same way we can project structure of items, threads with replies or combine it.

```
MATCH (g:GROUP) RETURN g LIMIT 10;
```



```
$ start u=node(15) match (t:THREAD)-[:BELONGS_TO]->(i:ITEM)<-[FOLLOWS]-(u),(r:REPLY)-[:CHILD_OF]->(t) return r,t order by r.updated,t.updated DESC skip 0 limit 50
```

```
CYPHER start u=node(15) match (t:THREAD)-[:BELONGS_TO]->(i:ITEM)<-[FOLLOWS]-(u),(r:REPLY)-[:CHILD_OF]->(t) return r,t order by r.updated,t.updated DESC skip 0 limit 50
```

✓ Displaying 12 nodes, 9 relationships

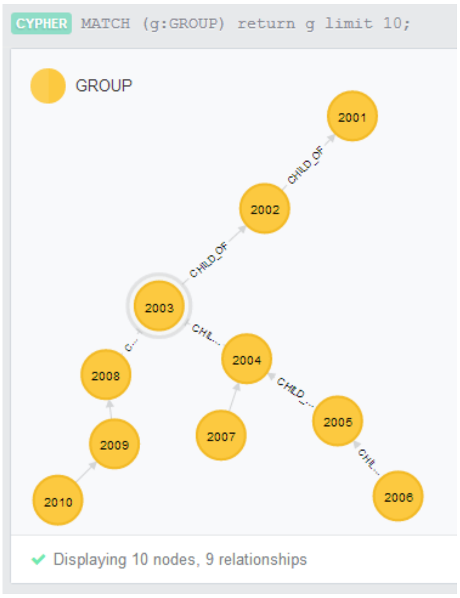
```
CYPHER start u=(15) match (t:THREAD)-[:BELONGS_TO]->(i:ITEM)<-[FOLLOWS]-(u),(r:REPLY)-[:CHILD_OF]->(t) return r,t order by r.updated,t.updated DESC skip 0 limit 50
```

```
Invalid input '(': expected whitespace, NODE or RELATIONSHIP (line 1, column 9)
"start u=(15) match (t:THREAD)-[:BELONGS_TO]->(i:ITEM)<-[FOLLOWS]-(u),(r:REPLY)-[:CHILD_OF]->(t) return r,t order by r.updated,t.updated DESC skip 0 limit 50"
         ^
```

⚠ Neo.ClientError.Statement.InvalidSyntax

```
PLAY :play welcome
```

Figure 4.1: A view of Neo4j web console — fullscreen. On top queries are entered and results are visualized in a stream in the rest of the display.



```
CYPHER MATCH (g:GROUP) return g limit 10;
```

✓ Displaying 10 nodes, 9 relationships

Figure 4.2: Example of groups in a hierarchy; our testing user is a member of group id 2003 (the user shown in the next figure.)

What is more interesting (and harder) is the task to find out, where the user has the *access privileges*. That is defined by the relationship `HAS_ACCESS`, which starts on user and ends on an item. You can see a partial visualization of the relevant testing data on fig. 4.3. Let me now introduce the part of the query, which finds the access rights:

```
START u=node(1000)
 MATCH (u:USER)-[:MEMBER_OF|:HAS_ACCESS*]->(direct:ITEM)
  WITH direct // sending further the directly accessible
  MATCH path=(direct)<-[:CHILD_OF*0..]-(accessible:ITEM)
   WHERE none(link IN tail(nodes(path))
              WHERE ()-[:HAS_ACCESS]->(link))
   RETURN accessible;
```

This query basically tries to find out, whether the user has access through a group privileges or their own privileges and also traverses the children, considering possible changes in access rights settings.

To explain this complex query in detail we must undergo a short lesson on a few advanced abstractions used in the *Cypher* language, which is very similar to the constructs in SQL, meaning that a reader familiar with that shouldn't find any difficulties.[2]

`WITH` can manipulate the data before passing to subquery (*Cypher* really does not allow subqueries in brackets as SQL does); in the easiest case it just selects what variables are passed.

`path=` is saving of the matched paths into collection.

`*` **or** `*0..` **or** `*1..5` similarly to regular expressions, it is repeating of the relationship 1–N times, or 0–N times. One has to be very careful not to get into infinite searches here, though.

`tail(list)` extracts the rest of a list, after the head is removed.

`all(p IN list WHERE condition)` is true if all of the conditions for members of the list are true, alternatively `none()` could be used with negated condition, which would have the same effect (even in terms of performance).

`|` is a logical *OR*, allowing to match different relationships.

[2]If you are missing any knowledge on *Cypher* query language, please refer to the Cypher Language Quick Overview, where the basics were explained (2.3.1 on page 16).

`NOT` is logical negation.

`AS` is the alias, can be also used to filter values of variables with filter functions.

So, what the above-mentioned query does precisely is the following (see figure 4.3 for visualization):

1. Start with the node ID 1000, save it as `u` (our user).
2. Find all nodes that this user has access/member of path of arbitrary length.
3. We are also interested in all the children of those nodes.
4. But neither these children, nor any nodes on the path from the node, where the user has direct access, should ever have a relation `HAS_ACCESS`. That would mean different access privileges.
5. Return all of these child nodes and direct access nodes, where the user has access, limit the number of results to 100.

Consideration of Further Optimizations

While researching the knowledge on how to optimize the recursive queries with *Cypher* I found many recommendations, so why not list some of them:

1. Don't pass unnecessary results onto subqueries with `WITH`. This recommendation is especially worthy, if the passed variables contain many nodes — if they are not necessary for matching, then they simply take precious memory space needlessly.
2. To avoid combinatorial explosion, use `shortestpath()`. Imagine using a match `(a)<-[r:REL*]-(b)`, when the graph has cycles. That would result in an infinite loop. But from definition, trees don't have cycles. Even so, when multiple paths are possible in dense graphs, all combinations of those would match. But fortunately again, trees are already kernels, meaning there can't be two paths between two nodes. In other words, this case can't happen in our problem domain. To be sure that it does not happen by mistake, relations have different labels and are explicitly named in matches.

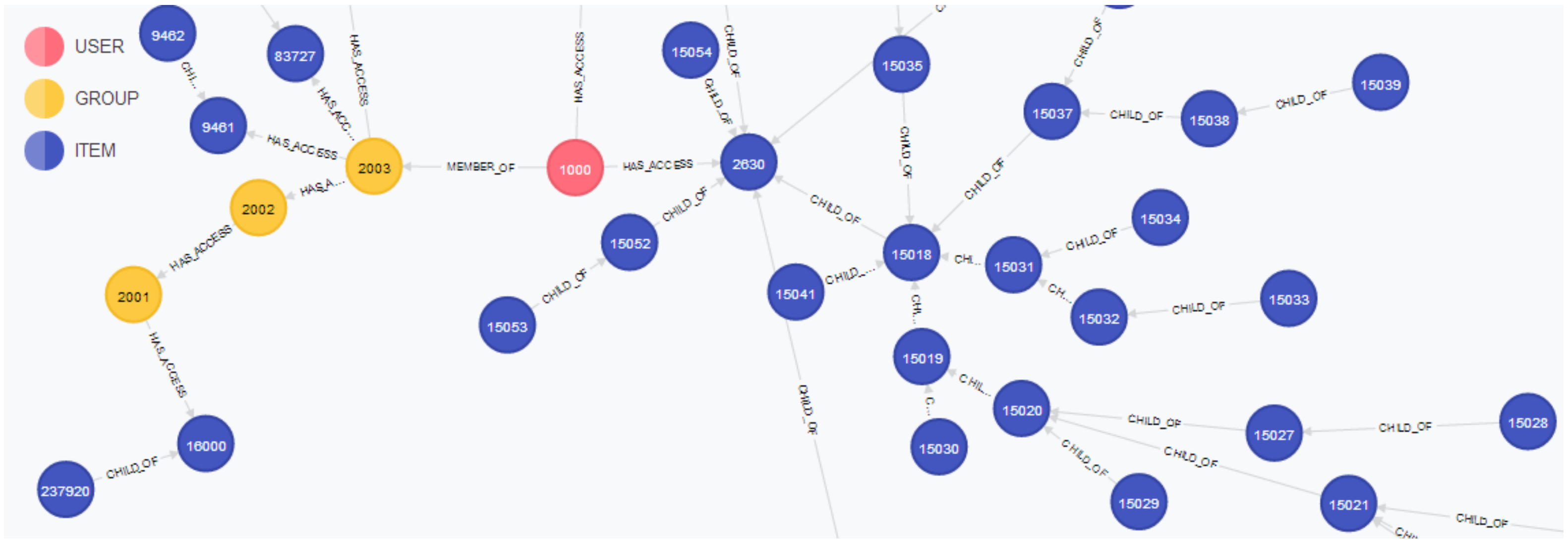
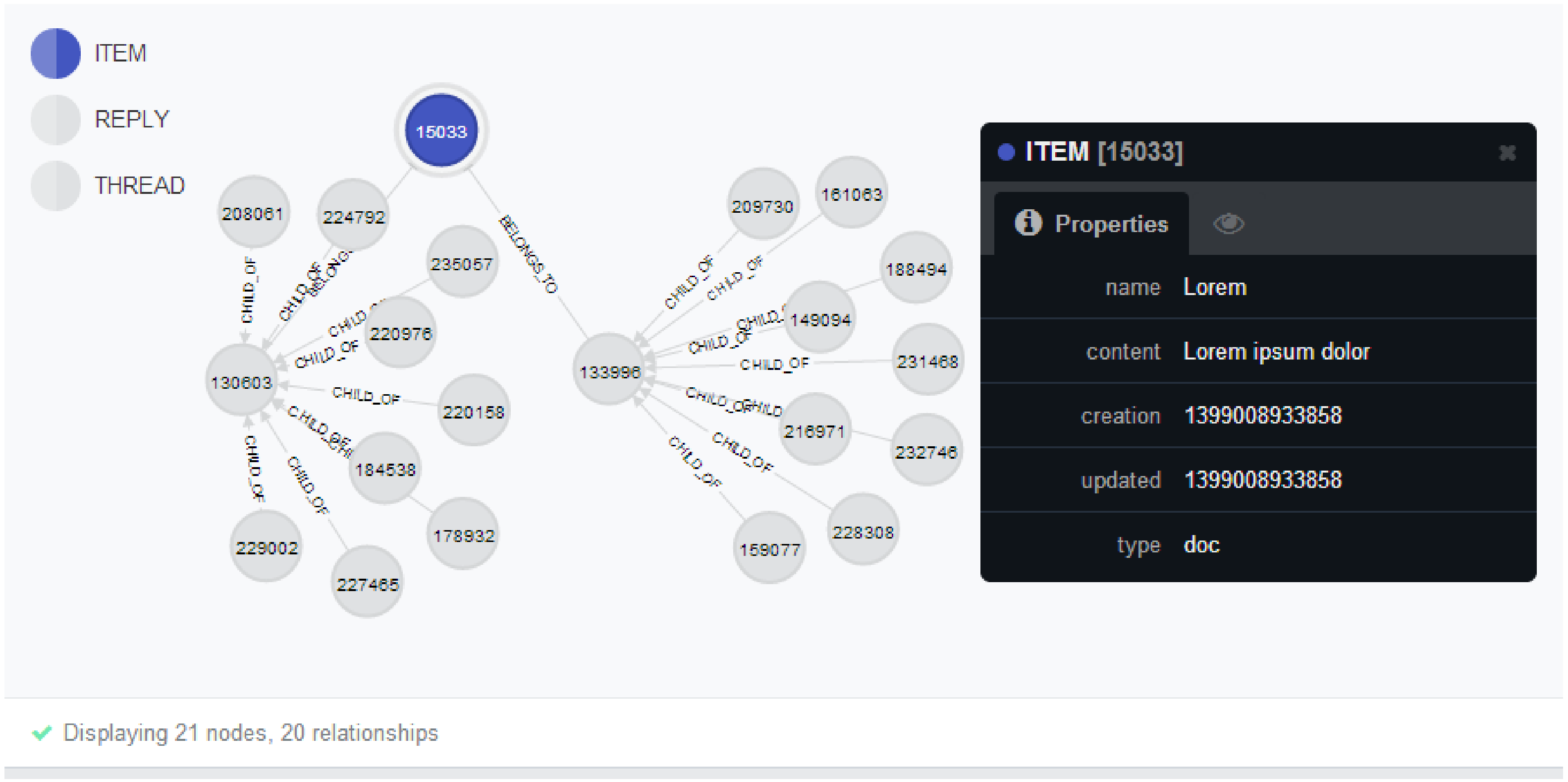


Figure 4.3: defining resolution of access rights is the hardest part. Here, we can see the user together with all the tree structure they can access. Also, notice the yellow groups — the user is member of a group, which is a subgroup of a higher group, which has access on an item with id 16000; therefore the user is able to access this item and its child item, but no further children, as this child has already different range of access rights effectively cutting off our example user.



● ITEM [15033]

Properties

name	Lorem
content	Lorem ipsum dolor
creation	1399008933858
updated	1399008933858
type	doc

✓ Displaying 21 nodes, 20 relationships

Figure 4.4: But what we are really interested in are the comments and threads, so once we know where we *do* have access, we might want to retrieve those. There is one `ITEM` with two `THREAD`s with multiple `REPL`ies in the picture.

3. If it is a constraint, put it into `WHERE`; if it should be returned, then into `MATCH`. This not only helps clear the query for easier reading by humans, but may have potentially positive effects on performance.

4. Don't use expensive operations as `DISTINCT` or `ORDER BY`. Well, unfortunately, we can't finish our task without ordering.

4.1.1 The Final Global Query

Now that we have seen, how to get the `ITEM`s, let us now skip to the final implementation of the global newsfeed in Neo4j:

```
START u=node(1000)
 MATCH (u:USER)-[:MEMBER_OF|:HAS_ACCESS*]->(direct:ITEM)
  WITH direct
  MATCH path=(direct)<-[:CHILD_OF*0..]-(accessible:ITEM)
   WHERE none(link IN tail(nodes(path))
             WHERE ()-[:HAS_ACCESS]->(link))
   WITH accessible
   // Here you can see that by specifying possible length of
    relationship as 0 we are practically merging THREADs and
    REPLies into one collection - comment.
   MATCH (accessible)<-[:BELONGS_TO]-(t:THREAD)
        <-[:CHILD_OF*0..1]-(comment)
   RETURN comment ORDER BY comment.updated skip 100 limit 50;
```

The new part in that code is the second one, where we are matching all the `THREAD`s of the accessible `ITEM`s, merging them with their `REPLies` and sorting by time they were updated. This way we get the final result — the newsfeed of comments.

Note that even though we are skipping and limiting, all the matching nodes from the query must be fetched and held in the database cache and then sorted. If we leave the sorting out, then the query is much faster, as we just receive a cursor on which we can iterate – the database could fetch the nodes lazily.

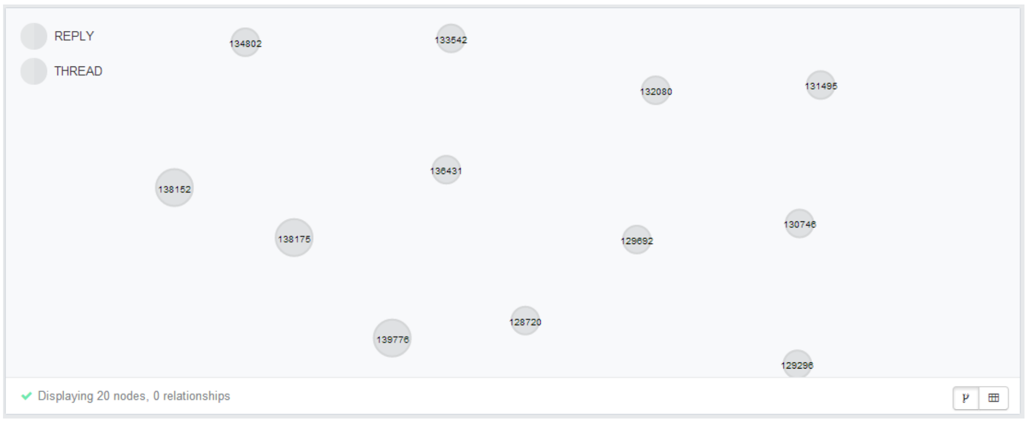
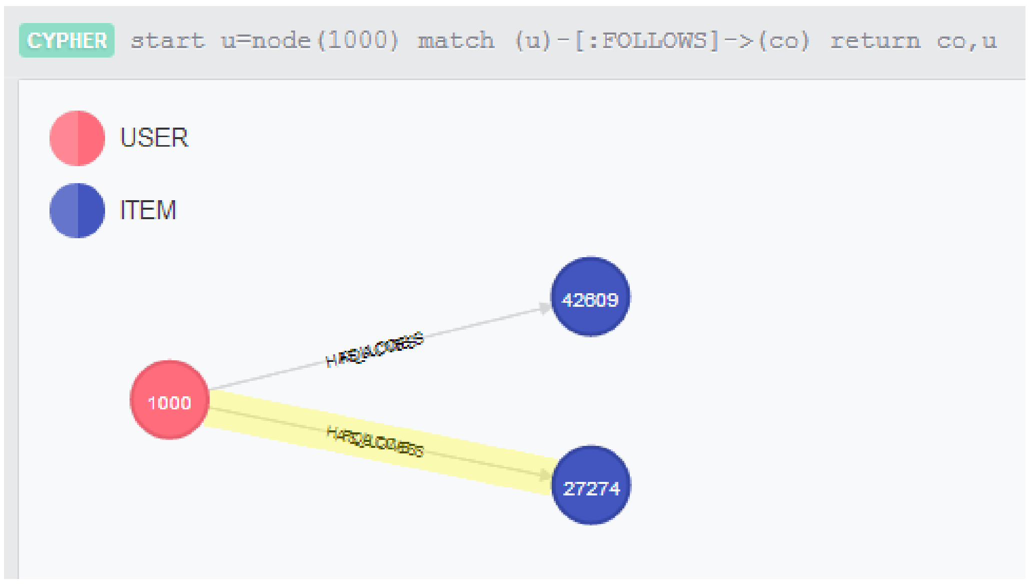


Figure 4.5: Final result — newest comments. does not actually look quite graph-like, because we are now retrieving only those separate nodes, and throwing away all the other relationships and the nodes we used on the way to find comments with the query.

4.1.2 Final Followed Query

```
START u=node(50)
 // Only followed items, please
 MATCH (u:USER)-[:FOLLOWS]->(followed:ITEM)
  WITH followed, u
  MATCH (u:USER)-[:MEMBER_OF|:HAS_ACCESS*]->(direct:ITEM)
   WITH direct, followed
   // Only accessible child nodes of followed, actually
   MATCH path=(direct)<-[:CHILD_OF*0..]-(followed)
              <-[:CHILD_OF*0..]-(interesting:ITEM)
    WHERE none(link IN tail(nodes(path))
               WHERE ()-[:HAS_ACCESS]->(link))
    WITH interesting
    MATCH (interesting)<-[:BELONGS_TO]-(t:THREAD)
          <-[:CHILD_OF*0..1]-(comment)
    RETURN comment ORDER BY comment.updated skip 100 limit 50;
```

In this last, but longest, *Cypher* query we already can recognize most of the previous code. The new part adds filtering by `FOLLOWS` relationship. Actually, it adds a starting point, just after we select the user, who wants the newsfeed, we check his followed items and continue from there on with checking all the children, not forgetting about the access rights. In reality we are doing a multilateral match on the nodes.



```
start u=node(1000) match (u)-[:FOLLOWS]->(co) return co,u
```

Figure 4.6: So far, we were considering only the global newsfeed, but part of the task is to fetch user-customized lists of updates on followed pages. To do that we also need to add the `FOLLOWS` relationship. It is not very well visible in the figure, because there is also the `HAS_ACCESS` relationship and the Neo4j web interface just overlays them, which is not very user friendly.

Indexing

It is worth mentioning that an index was created on the updated property of both `THREAD` and `REPLY` before running the queries to test performance. Supposedly, it should enhance performance on the sorting operation, but it is not clear, whether the database really makes use of it.

Additionally, the performance was compared with a model that changed all `THREAD` and `REPLY` nodes to `COMMENT` label and set index on this one instead; no improvement was noticed, however, so the original solution was kept.[3]

```
CREATE INDEX ON :REPLY(updated);
```

Note on Database Profiling

For MySQL there exists the `EXPLAIN` command, which reveals the execution plan to the advanced database user for optimizations (see section 5.3). Similar functionality is unfortunately not so easy to find and is not mentioned in the standard courses of Cypher. Yet, it exists. But even members of the Neo4j team stated publicly that it is not well suited for usage at the moment and that the interface is not very user friendly. Nevertheless, it might be useful for fine-tuning queries.

In the figure 4.8, the first part just simply lists the items contained in the result set. The second part, let us call it the "call stack", contains the execution plan; ie. stack of functions used by the engine. It is very complicated and mostly unexplained anywhere in detail as far as I know. But one can for example use the information about how much the query hits the data in certain parts and maybe decide to rewrite it, so that it hits less, resulting in less reads from permanent storage.

4.2 Neo4j BatchInserter

Since we needed to generate and save a lot of nodes fast[4], the previously introduced manner with creating nodes through template using automatic mapping could not be used, as it took too much time.

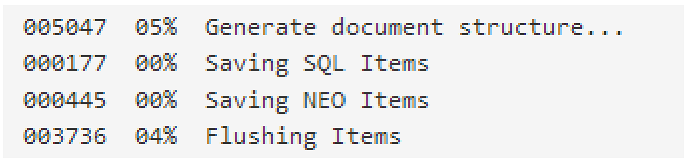
But Neo4j embedded can be run in `BatchInserter` mode which does not consider transactions and has several restrictions. At the same time, however, it allows saving the nodes orders of magnitude faster than the normal approach.

In the following snippet one can see how easily the `BatchInserter` is used for both node and relationship cretion. If a relationship is being created, the end nodes already have to exist, otherwise an exception is thrown.

```
public void neoBatchInsert(Group[] groups) {
  // gdbm is my proxy that manages database instances
  BatchInserter i = gdbm.getBatchInserter();
  Map<String, Object> mapping;
  for (Group g : groups) {
    mapping = MapUtil.map("name", g.getName());
    i.createNode(g.getId(), mapping, Labels.GROUP);
    // null value => no relationship
    if (g.getParent() != null) {
      i.createRelationship(g.getId(), g.getParent(), Relationships
    .HAS_ACCESS, null);
    }
  }
}
```

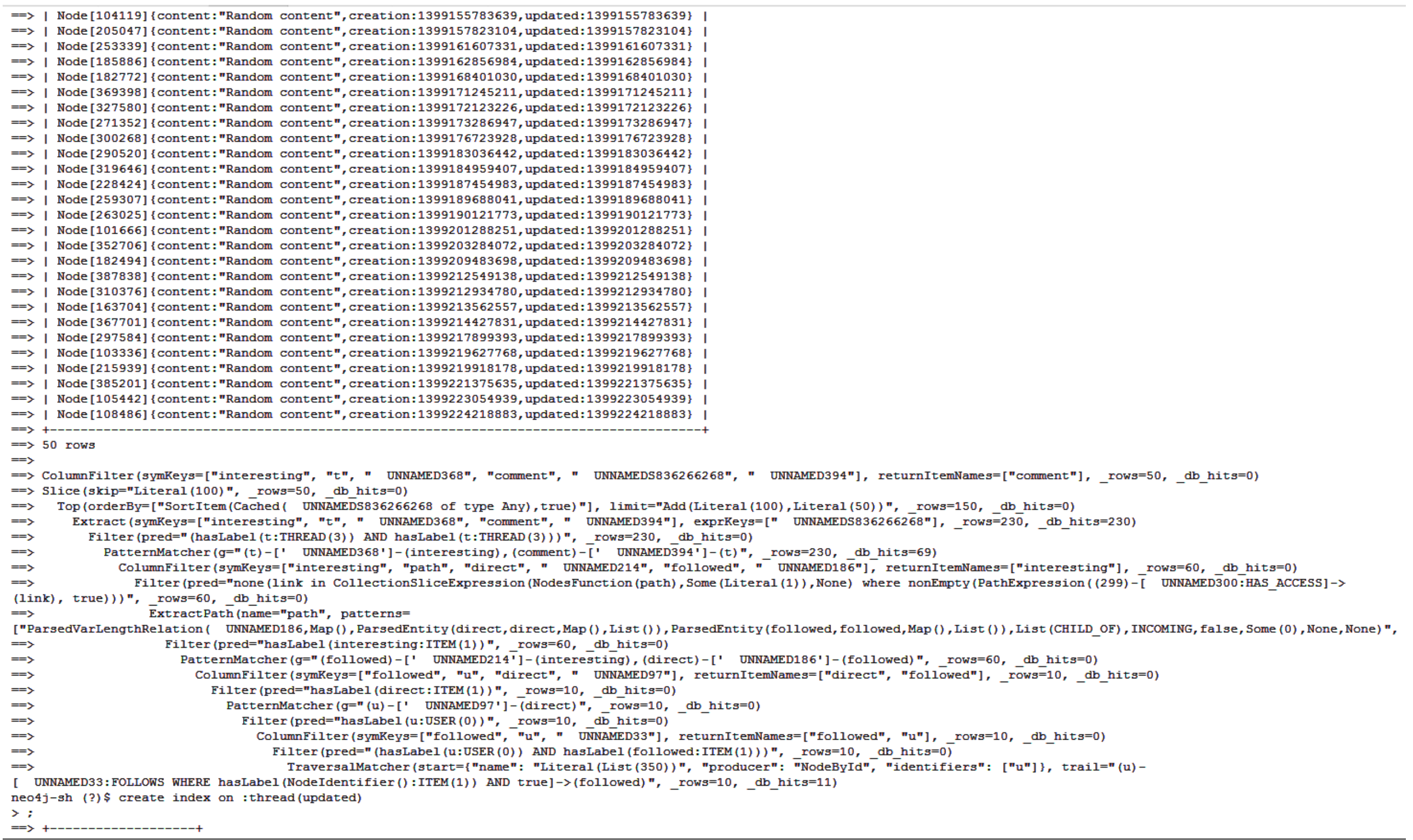
[3]There is also a possibility to use so called legacy indexing, but not through Cypher. As it did not seem as a standard feature it was not used for the testing. More on this you can find in the documentation [Neo4j(2014)].

[4]For more information on fast importing data into Neo4j, see this web address: http://www.neo4j.org/develop/import



```
005047  05%  Generate document structure...
000177  00%  Saving SQL Items
000445  00%  Saving NEO Items
003736  04%  Flushing Items
```

Figure 4.7: Comparison of time taken (in milliseconds) to save 20 000 items ("flushing" is importing CSV into the MySQL database).



```
==> | Node[104119]{content:"Random content",creation:1399155783639,updated:1399155783639} |
==> | Node[205047]{content:"Random content",creation:1399157823104,updated:1399157823104} |
==> | Node[253339]{content:"Random content",creation:1399161607331,updated:1399161607331} |
==> | Node[185886]{content:"Random content",creation:1399162856984,updated:1399162856984} |
==> | Node[182772]{content:"Random content",creation:1399168401030,updated:1399168401030} |
==> | Node[369398]{content:"Random content",creation:1399171245211,updated:1399171245211} |
==> | Node[327580]{content:"Random content",creation:1399172123226,updated:1399172123226} |
==> | Node[271352]{content:"Random content",creation:1399173286947,updated:1399173286947} |
==> | Node[300268]{content:"Random content",creation:1399176723928,updated:1399176723928} |
==> | Node[290520]{content:"Random content",creation:1399183036442,updated:1399183036442} |
==> | Node[319646]{content:"Random content",creation:1399184959407,updated:1399184959407} |
==> | Node[228424]{content:"Random content",creation:1399187454983,updated:1399187454983} |
==> | Node[259307]{content:"Random content",creation:1399189688041,updated:1399189688041} |
==> | Node[263025]{content:"Random content",creation:1399190121773,updated:1399190121773} |
==> | Node[101666]{content:"Random content",creation:1399201288251,updated:1399201288251} |
==> | Node[352706]{content:"Random content",creation:1399203284072,updated:1399203284072} |
==> | Node[182494]{content:"Random content",creation:1399209483698,updated:1399209483698} |
==> | Node[387838]{content:"Random content",creation:1399212549138,updated:1399212549138} |
==> | Node[310376]{content:"Random content",creation:1399212934780,updated:1399212934780} |
==> | Node[163704]{content:"Random content",creation:1399213562557,updated:1399213562557} |
==> | Node[367701]{content:"Random content",creation:1399214427831,updated:1399214427831} |
==> | Node[297584]{content:"Random content",creation:1399217899393,updated:1399217899393} |
==> | Node[103336]{content:"Random content",creation:1399219627768,updated:1399219627768} |
==> | Node[215939]{content:"Random content",creation:1399219918178,updated:1399219918178} |
==> | Node[385201]{content:"Random content",creation:1399221375635,updated:1399221375635} |
==> | Node[105442]{content:"Random content",creation:1399223054939,updated:1399223054939} |
==> | Node[108486]{content:"Random content",creation:1399224218883,updated:1399224218883} |
==> +------------------------------------------------------------------------------------+
==> 50 rows
==>
==> ColumnFilter(symKeys=["interesting", "t", "  UNNAMED368", "comment", "  UNNAMEDS836266268", "  UNNAMED394"], returnItemNames=["comment"], _rows=50, _db_hits=0)
==> Slice(skip="Literal(100)", _rows=50, _db_hits=0)
==>   Top(orderBy=["SortItem(Cached(  UNNAMEDS836266268 of type Any),true)"], limit="Add(Literal(100),Literal(50))", _rows=150, _db_hits=0)
==>     Extract(symKeys=["interesting", "t", "  UNNAMED368", "comment", "  UNNAMED394"], exprKeys=["  UNNAMEDS836266268"], _rows=230, _db_hits=230)
==>       Filter(pred="(hasLabel(t:THREAD(3)) AND hasLabel(t:THREAD(3)))", _rows=230, _db_hits=0)
==>         PatternMatcher(g="(t)-['  UNNAMED368']-(interesting),(comment)-['  UNNAMED394']-(t)", _rows=230, _db_hits=69)
==>           ColumnFilter(symKeys=["interesting", "path", "direct", "  UNNAMED214", "followed", "  UNNAMED186"], returnItemNames=["interesting"], _rows=60, _db_hits=0)
==>             Filter(pred="none(link in CollectionSliceExpression(NodesFunction(path),Some(Literal(1)),None) where nonEmpty(PathExpression((299)-[  UNNAMED300:HAS_ACCESS]->
(link), true)))", _rows=60, _db_hits=0)
==>               ExtractPath(name="path", patterns=
["ParsedVarLengthRelation(  UNNAMED186,Map(),ParsedEntity(direct,direct,Map(),List()),ParsedEntity(followed,followed,Map(),List()),List(CHILD_OF),INCOMING,false,Some(0),None,None)",
==>                 Filter(pred="hasLabel(interesting:ITEM(1))", _rows=60, _db_hits=0)
==>                   PatternMatcher(g="(followed)-['  UNNAMED214']-(interesting),(direct)-['  UNNAMED186']-(followed)", _rows=60, _db_hits=0)
==>                     ColumnFilter(symKeys=["followed", "u", "direct", "  UNNAMED97"], returnItemNames=["direct", "followed"], _rows=10, _db_hits=0)
==>                       Filter(pred="hasLabel(direct:ITEM(1))", _rows=10, _db_hits=0)
==>                         PatternMatcher(g="(u)-['  UNNAMED97']-(direct)", _rows=10, _db_hits=0)
==>                           Filter(pred="hasLabel(u:USER(0))", _rows=10, _db_hits=0)
==>                             ColumnFilter(symKeys=["followed", "u", "  UNNAMED33"], returnItemNames=["followed", "u"], _rows=10, _db_hits=0)
==>                               Filter(pred="(hasLabel(u:USER(0)) AND hasLabel(followed:ITEM(1)))", _rows=10, _db_hits=0)
==>                                 TraversalMatcher(start={"name": "Literal(List(350))", "producer": "NodeById", "identifiers": ["u"]}, trail="(u)-
[  UNNAMED33:FOLLOWS WHERE hasLabel(NodeIdentifier():ITEM(1)) AND true]->(followed)", _rows=10, _db_hits=11)
neo4j-sh (?)$ create index on :thread(updated)
> ;
==> +-------------------+
```

Figure 4.8: The deprecated webadmin console interface provides a feature called PROFILE. As with EXPLAIN one simply runs their query with this key word in front of it to get a listing similar to the above depicted one. (Screenshot was inverted to better fit the printed medium.)

As you have seen in the figure 5.1, the saving is really fast. The ”flushing” actually runs `LOAD DATA INFILE`, so one should add that to MySQL performance (this loading mechanism will be explained in section 5.4 on page 56). Also, worth mentioning is that indexing is run on `BatchInserter.shutdown()`, although we do not have time measurements for that, it did not take too long. The complete generation time also contains around 1 second that the tree generator takes, because it uses random number generator and that is a relatively (but linearly) costly operation

The best advantage of this API is that it saves even 10 times faster than the `LOAD DATA INFILE` instruction that was later used for SQL. Relative disadvantages, which we have to be aware of:

1. If the inserter is not `shutdown()` properly, then the database is corrupted.
2. It is not thread-safe; only one thread should use the inserter for one database at a time.
3. For maximum performance one has to be careful about memory and cache settings.

Another possible API option is the `BatchDatabase`, which has similar features as the inserter and keeps the normal database API; according to the documentation it reaches lower performance, however.

As a standalone alternative one can use an application available that reads data from Comma-Separated Values (CSV) files and uploads the respective nodes and relationships very fast. See [Hunger(2014)]. In the online article [Hunger(2012b)] the author states performance of 1 million nodes imported per second. But his approach involves different API and parallelization.

5 MySQL

In the chapter about *MySQL* we will see, how the database was used for the task at hand. Mostly, we will explain the same things we went through in the previous chapter, just with another database and language. The exception are the stored procedures, which were not used for Neo4j as it was embedded.[1]

Note on Primary Keys

There are two basic approaches to setting primary keys on tables in SQL; either the database designer utilizes some distinctive column that has a unique value for each row, or you can use a combination of such columns, which are unique only together as the combination. The other approach is to use an artificial primary key, an auto-incremented integer key.

The first approach is clearer in terms of design and minimization, but the second is often more practical. For this model a combination was used. For the tables that represent the same entities as nodes in the graph database an id was used. For tables representing many-to-many relations a composite primary key made from the two foreign keys was used, because a special id would be absolutely redundant in this case. This composite key creates also indexes on both columns, so they can be used later for `SELECT` conditions.

But even where ids were used, they could not be auto-incremental. That is because while using two database types at once the interoperability of the data is important. In the beginning of the development, there were auto increment primary keys on all the tables. But that meant that the keys were unique only per table. On the other hand, while importing into Neo4j, that database keeps node ids unique over the whole database. A solution would either be to add the per-table unique id as a property in Neo4j or to switch off auto-incrementation in MySQL and generate the indexes manually. The latter was chosen due to performance reasons, because the previously explained `BatchImporter` uses node ids for creating relationships.

[1]But there actually are two possibilities to do it even in Neo4j, though not with Cypher, analogically to SQL. One can use either Java API for RESTful Web Services (JAX-RS), which deploys Java Virtual Machine (JVM) (not only Java) code to the server; or as mentioned before writing your own layer above the embedded server enriching REST endpoints.

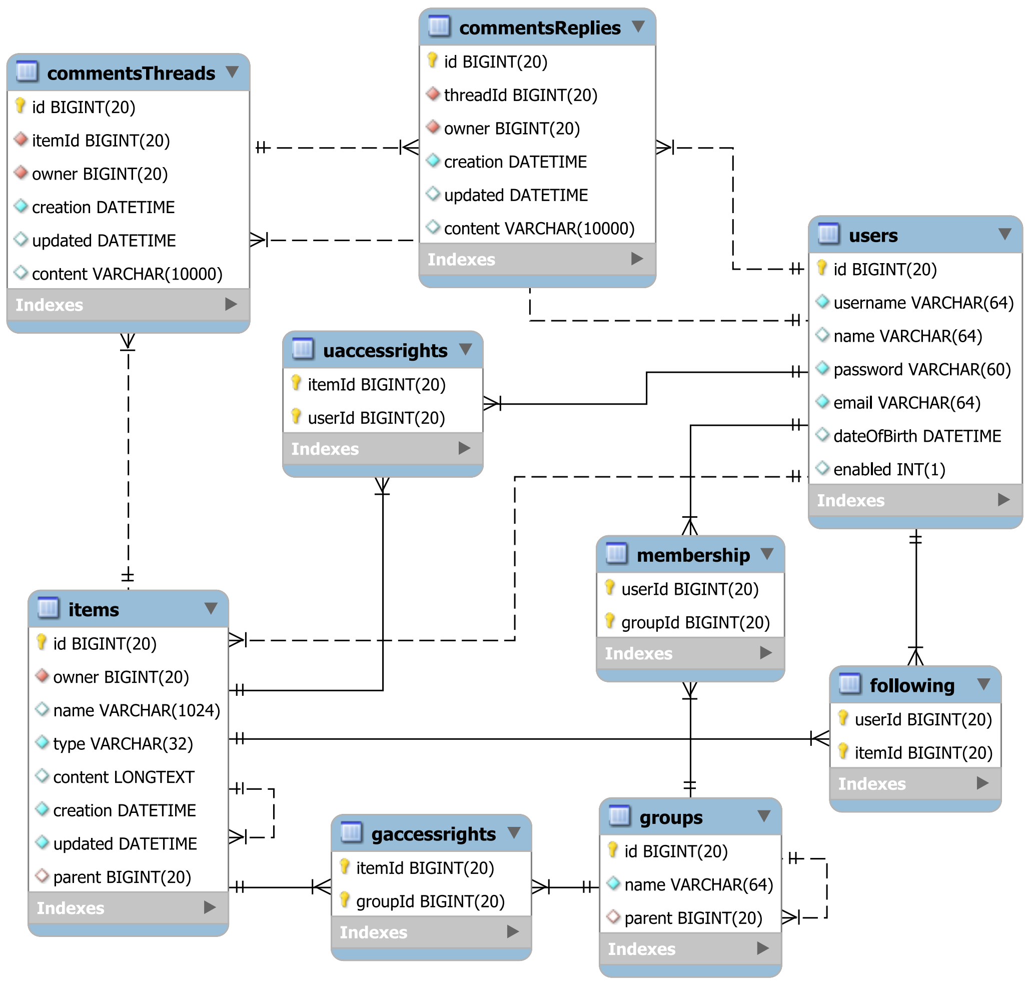


Figure 5.1: The above depicted model contains all the SQL tables with many-to-many tables as well. The relations are connecting the tables, not the exact columns, so in this respect the diagram can be a bit confusing. The little keys stand for primary keys. The red dots highlight the foreign keys. Empty red dots mean the foreign key can be null. The relations ||—< are one-to-many.

Tables

The table structure was created to be similar to the original application, but not completely the same. Simplifications were made, as we needed to study only a certain part of the application and did not need support for other features. Mostly, the model is self-explanatory.

commentsThreads contain the topics of discussion. There was a possibility to split the common parts with the replies into a common comment table and use inheritance (as it is done in the original application). The simpler model was used, though.

commentsReplies can only exist under threads.

groups are interesting, because they contain a recursive relation, which creates the hierarchical tree structure. Note that SQL is general enough to allows this, but does not have any constraints which could be used to check that cycles exist, so such problems would have to be detected by the programmer.

items are also connected in a hierarchical structure. In the model

users represents our virtual users; would be used for login etc.

Many-to-many Tables

These tables represent relations, which can not be expressed as foreign keys (one-to-one or one-to-many). Using the both foreign keys as the primary key, which must be unique immediately gives us certainty that there will not be any duplicity in these relations, which could have been created by the generator, which generates the data pseudo-randomly.

following can exist between the user and an item.

gaccessrights add access right for an item to a group.

membership of a user in a group.

uaccessrights add access right for an item to a user; it can not be merged with `gaccessrights` as the foreign key references a different table.

Indexes

Indexes are very important to speed up the queries above any model. They lower the computational complexity — it is not necessary to traverse the whole table and read all values to find a certain row when we have an index on the requested column — a structure that can be searched very quickly (as for example a hash map) containing direct addresses on the rows. It also should make sorting of a table faster, although it is unclear, whether it works while sorting results of a `UNION`, too.

All the indexes that were necessary for the queries were actually already created, because they were part of a foreign or primary key.

5.1 Stored Procedures

To lower the traffic between the database and the application, two "recursive" querying procedures were moved to the SQL database.[2] This traffic would grow with complexity of $O(n \cdot k)$, where n is the number of rows necessary for the current traversal and k is the constant time per returning the result set over the network.

Note that while declaring procedures through a client like MySQL Workbench, you have to use the command `DELIMITER` to change the delimiter of queries, so that the semicolons are not interpreted as the end of the procedure. While using JDBC or similar connectivity that is not necessary[3], because one can directly send the procedure as a whole.

The algorithm is actually a cycle; it was transformed from the recursive one, so that it was not necessary to change the database settings of the maximum recursive stack. Also, we theoretically will never know, how many cycles it will actually take to reach the closest ancestor group with access privileges, so setting a hard boundary is tricky. The stop condition is the `null` value of the `parent` column, which denotes the root group. If there were a cycle in the data, the procedure would be caught in an indefinite loop.

[2]We could use `WITH RECURSIVE` instead, if that was supported by *MySQL*, see theoretical section 2.1.1.

[3]And not even possible, since the `DELIMITER` command is only valid on the client like *MySQL Workbench* or `mysql` in the command line.

getGroupAncestors

In this procedure we want to fetch all the IDs of the transitive closure made from ancestors of a group. One should not forget to define what should happen if no row is found for a variable. A good choice is setting it to null instead of nothing happening, which is the default behavior that can lead to hidden bugs.

```
CREATE PROCEDURE `getGroupAncestors`(IN gi BIGINT(20))
  BEGIN
    DECLARE cur BIGINT(20); -- Create local variable
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET cur = NULL;
    SET cur = gi;
    WHILE cur IS NOT NULL DO
      -- Find the parent of currently selected group.
      SELECT g.parent INTO cur FROM groups AS g WHERE id = cur;
      -- If the parent is not null, return it.
      IF cur IS NOT NULL THEN
        SELECT cur;
      END IF;
    END WHILE;
  END;
```

getParWithAccessRights

```
CREATE PROCEDURE `getParWithAccessRights`(IN itId BIGINT(20))
  procedure_label:BEGIN -- Labeling enables premature exit
    DECLARE cur BIGINT(20);
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET cur = NULL;
    SET cur = itId;
    WHILE cur IS NOT NULL DO
      IF EXISTS (SELECT 1 FROM gaccessrights AS gar WHERE gar.
  itemId = cur) THEN
        SELECT cur; -- Once we have the node, we can return
        LEAVE procedure_label;
      ELSEIF EXISTS (SELECT 1 FROM uaccessrights AS uar WHERE uar.
  itemId = itId) THEN
        SELECT cur; -- The same with user access rights.
        LEAVE procedure_label;
      END IF; -- Iterate up the Item tree structure
      SELECT i.parent INTO cur FROM items AS i WHERE i.id = cur;
    END WHILE;
  END
```

5.2 Queries for the Newsfeed

Let us begin with the explanation of the global newsfeed fetching, as it is simpler and without the constraints on the ”follow” relation, which will be explained shortly as well.

We begin with selecting all the comments from all the items, ordering them by the ”updated” column:

```sql
SELECT t.id, t.updated, t.itemId, t.content, t.owner, t.creation
  FROM commentsthreads as t
  -- Merge the results. The columns must be the same
  UNION ALL (SELECT
     r.id, r.updated, t.itemId, r.content, r.owner, r.creation
     FROM commentsreplies AS r
     INNER JOIN commentsthreads AS t ON r.threadId = t.id)
  ORDER BY updated;
```

Now, we can do a short trip to Java and JDBC, as it is important to mention that we probably do not want to fetch all the comments at once, which could be thousands to millions records. Instead we set the parameters accordingly, so the data is sent only once it is needed (lazy cursor approach):

```java
PreparedStatement ps = conn.prepareStatement(sqlString,
   ResultSet.TYPE_FORWARD_ONLY, // important as well
   ResultSet.CONCUR_READ_ONLY)
/* This constant should be set to something reasonable, like a 100
    or 1000. If it is too low it can be lowering performance as
    well. */
ps.setFetchSize(FETCH_SIZE_LIMIT);
```

Having all the possible updates sorted, we have to check, where the user has the access right, we definitely do not want to show anything that the user should not see. As it is possibly very costly, we are running it as the secondary filter. There are three types of access rights possible:

1. direct privileges granted to the user,
2. privileges granted to a group that the user is a member of,
3. privileges of ancestor groups of the group from previous point.

We can begin with the first case (it is a basic `SELECT`):

```sql
SELECT uar.itemId FROM uaccessrights AS uar WHERE uar.userId = ?
```

For the second and third case, we will select all the groups that the user belongs to. We are not selecting their access rights immediately (with a `JOIN`), because we need above all their IDs to be later able to traverse the group structure to find all their ancestors:

```sql
SELECT m.groupId FROM membership AS m WHERE m.userId = ?;

 -- using the group IDs we got from the previous query:
CALL getGroupAncestors(?)
```

Now, we can merge all the group IDs, which provide some access rights to the user and prepare an SQL query that will give us all the directly accessible items.

```sql
SELECT gar.itemId FROM gaccessrights AS gar WHERE gar.groupId IN
    (?, ?, ?, ...)
```

Notice, that there have to be as many question marks as we got unique IDs, from which we want to fetch the relations. This requires dynamic building of the SQL query and prohibits its caching (of the compiled code). Unfortunately, because it can be each time totally different from the point of view of the compiler.

The other possible solution would be using singular queries, but that is even less efficient, although they can be sent at once. Another solution that could occur to us is batching the queries. But unfortunately that is only possible for execution of statements that are meant to change the state of the database. Statements that return a result set are not allowed to be batched.

So after this preparations we know, where the user has a direct access and we also have a cursor on a long list with all the comments sorted, we can start browsing them and checking the access rights. We do that by looking up the closest ancestor with access right settings in the tree structure. That is exactly what our previously defined procedure does, so we can just call it with `CALL getParWithAccessRights(?)`.

Next, we check if the ancestor of the item to which the current comment belongs to is contained in the set of directly accessible items we already have. If yes, we add it to the result and continue checking the next one, if we do not have enough yet, once we do or there are no more comments, we stop.

Followed Filtering

So that was the algorithm behind the global newsfeed, the followed one adds more pre-filtering of the list, so we receive updates only on those pages that are accessible and at the same time followed by the user.

First, let us find out, what the user follows:

```sql
SELECT f.itemId FROM following as f WHERE f.userId = ?;
```

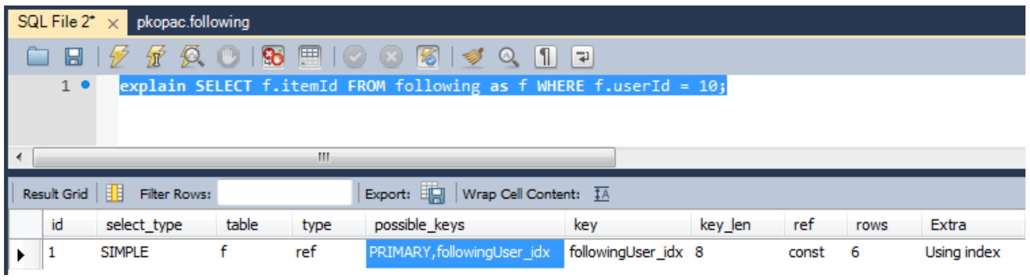
That was too easy maybe — we nearly forgot about all the child items; but how to recursively add all of the children (again a transitive closure)? This time we can use the BFS algorithm to lower the number of queries to the maximum of the depth of the subtree per followed item (a similar approach is taken in the original application):

```sql
SELECT i.id FROM items as i WHERE i.parent IN (?, ?, ?...)
```

We should run this for each level, using the fetched IDs as new arguments and storing all of them merged as well until we will not get any new IDs. Then we can merge all the IDs of the subtrees.

One could also think about the problem with too long queries, but this problem is quite still far as the maximum default length on MySQL is 1 MB and by configuration it is actually possible to maximize it up to 1 GB using `max_allowed_packet`. [Oracle(2014)]

So now that we have all the interesting items, we will fetch all the comments, but this time with a `WHERE` condition on the Ids. As you can see the condition is actually inserted twice — on the first query and the `JOIN` inside the second. According to the rule: the less that is fetched, the better. Repeating the same condition in itself is not an issue.



```
explain SELECT f.itemId FROM following as f WHERE f.userId = 10;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	f	ref	PRIMARY,followingUser_idx	followingUser_idx	8	const	6	Using index

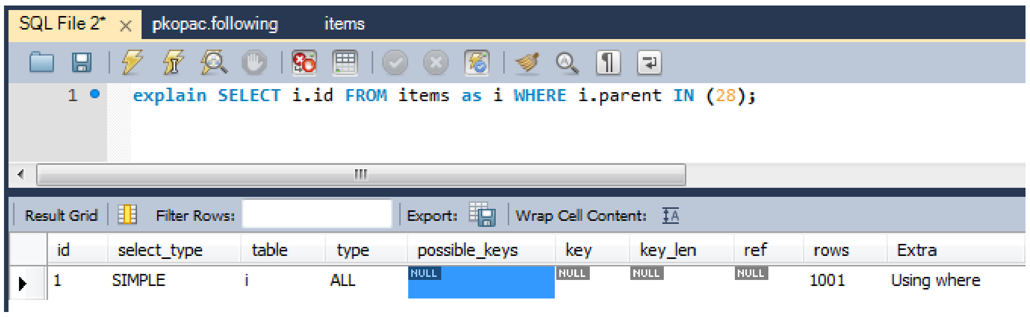
Figure 5.2: In this figure we can see that *MySQL* does not have available index and has to resort to row-by-row comparison, which results in 1001 hits.

```
SELECT t.id, t.updated, t.itemId, t.content, t.owner, t.creation
  FROM commentsthreads as t
  -- This is the condition
  WHERE itemId IN (?, ?, ?...)
  UNION ALL (SELECT
     r.id, r.updated, t.itemId, r.content, r.owner, r.creation
     FROM commentsreplies AS r
     INNER JOIN commentsthreads AS t ON r.threadId = t.id
     -- And again for the seconds part:
     WHERE itemId IN (?, ?, ?...))
  ORDER BY updated;
```

The rest of the process is the same as with the global newsfeed. Actually, it is only one method with a boolean switch in the implementation.

5.3 Profiling with `EXPLAIN`

MySQL offers a method to research the execution plan of the query, it is similar in usage to `PROFILE` from *Neo4j*, but much easier — one can run it in a standard client as the Workbench or Command Line Interface (CLI) `mysql`. Also, it is more user-friendly, because the results are server in a neat form, see fig. 5.2 and 5.3. After we run all our queries through `EXPLAIN` we will have insight in possibilities of speeding up `JOIN`s, conditions and sorting.



```
explain SELECT i.id FROM items as i WHERE i.parent IN (28);
```

```
id  select_type  table  type  possible_keys  key   key_len  ref   rows  Extra
1   SIMPLE       i      ALL   NULL           NULL  NULL     NULL  1001  Using where
```

Figure 5.3: In this case the query is similar to the one from the previous figure, but this time there is already defined a useful index, which can be used and therefore only 6 rows are hit.

5.4 MySQL `LOAD DATA INFILE`

Firstly, during the first iterations of development the application was using the normal `INSERT`ing process. But that is actually not the most efficient way how to import data into the *MySQL* database.

Much better approach is to save the data on the hard drive into CSV files and then after all is generated, this special instruction is called and the database loads the file:

```
LOAD DATA INFILE ? INTO TABLE items FIELDS TERMINATED BY ','
    ENCLOSED BY '\"' (id, owner, name, type, content, creation,
    updated, parent);
```

It is important that the database can access the file and it should be an absolute path on the file system. Also notice, that the command has a similar structure as `INSERT`, naming the order of the columns as they are saved in the CSV file, but it also has some features related to format of the data.

This can be up to 20 times faster than the normal `INSERT` as the official *MySQL* documentation claims. It is the best option that I found for such a bulk import and surprisingly the doubled writing to the drive is not that slow. See the comparison of times on figure 5.1 on page 48, where there are times of writing to the CSV file and then flushing into the SQL database compared to the time taken by `BatchInserter`.

6 Object-oriented Design

The project was based on Object-Oriented Programming (OOP). As Java, which is the implementation platform, is an object-oriented language, this is the natural approach. There were used nearly all the elements of this design that one can think of and the concrete utilization will be described in the following sections shortly. We also should not forget about multiple design patterns typical for OOP; those will be listed and explained only superficially, for in detail description you can refer to specialized literature like [Pecinovský(2007)] or [Metsker(2006)].

6.1 Domain Objects

For the representation of the reality of the application and its data we model each different element type as a class. The relations between the elements could be modeled as properties of the elements or as virtual elements holding the IDs of the endpoints. The second possibility was chosen, because it is closer to the representation in SQL tables and makes it easier (more efficient) to bulk insert data. The list of DO classes follows:

Package cz.pkopac.thesis.db

- Following
- GAccessRight
- Group
- Item
- Membership
- Reply
- Thread
- UAccessRight
- User

6.2 Data Access Objects

Repositories or in other words DAO are the logic inside the data layer. The pattern dictates they make public methods to do individual tasks like `Dog findDogByID(int id)` or `createNewHouse(House h)`. In the performance testing application there are three types of DAOs:

- batch insertion,
- newsfeed fetching,
- database operations.

Their architecture and function will be explained in the following pages to familiarize the reader with the design of the data layer of the project. All the DAOs that should be used in the application have their respective interfaces, and should be used only through those. Then, when there is a need to change the DAO for a different implementation, it is very easy. The interfaces are located in the package: `cz.pkopac.thesis.testing.services`

Batch Insertion

There are several classes used for fast insertion; first, we will describe the main interface. It is relatively small with just one method, but is generic, so that it can work with any domain object:

```
public interface ITBatchService<T> {
  public void batchInsert(T[] list, boolean flushing);
}
```

The implementing method is supposed to insert the objects from the list into the databases that are used by the project. The parameter `flushing` denotes, whether it is the last insert (that is important to know, because we will need to flush the CSV files into SQL database in the implementation).

This is a general interface and there are inheriting ones for each domain object. Those actually do not even need to do anything more than just specify the DO.

```
public interface ITGroupService  extends ITBatchService<Group>{

}
```

Having a special interface for each DAO service is important for the inversion of control pattern, where we define the the DAO as a `@Repository` implementing certain interface and then the container is able to auto-wire properly these DAOs wherever they are needed. The following support classes are placed in the package `cz.pkopac.thesis.testing.dao.support`.

For the batch inserting the class `DBBuffer<T>` was added, where the objects are stored in a queue and then after one workload unit is prepared, sent to the batching DAO. It manages flushing/committing of the inserted objects with the DAOs.[1]

To program the individual inserting DAOs, which have all similar function, the common code was extracted from the DAOs to upper classes. The class `TAbstractParallelDAO` implements simple consumer-producer parallelization pattern and leaves the logic to be programmed by inheriting classes. Saving to SQL runs in the current thread and to Neo4j in a daemon thread.[2] We synchronize at the end of each workload unit. This class also handles counting time for profiling, how long saving in each database takes.

The class `TCSVParalelDAO` extends the previous and adds the functionality of saving data into CSV files. Then, when flushing the DAO, the CSV data is saved into the database.

Before it was switched to the `TCSVParalelDAO` design, the application used SQL `SimpleJdbcInserter` — for that the DAOs used a class implementing the insertions called `TGeneralDAO`, capabilities of which were used through composition pattern.

While using all this support only the distinctive logic associated with the concrete domain object class stays in the final DAOs:

[1]It would be possible to make this more efficient by using thread safe queues, but when the interface was designed, `BatchStatements` for SQL were used, which require to set all the batch data first and then execute it once after a certain amount.

[2]I did not parallelize with the generator logic, even though that would be possible, because the most time-consuming operations are the I/O-related ones and those had to be done in parallel threads.

```java
@Override
protected void sqlBatchInsert(Item[] items) {
  for (Item i : items) {
    getCsvTmpWriter().addRecord(i.get...);
  }
}
```

The method `getCsvTmpWriter()` used in the above code returns an object, which manages saving a CSV file for the SQL and is prepared by the parent class. For Neo4j only the transformation needed for the `BatchInserter` is needed to be written:

```java
@Override
public void neoBatchInsert(Item[] items) {
  BatchInserter i = gdbm.getBatchInserter();
  Map<String, Object> mapping;

  for (Item it : items) {
    // 1) prepare mapping of properties
    // 2) create node
    // 3) create any relationships that are needed
  }
}
```

Newsfeed Fetching

The common interface is `INewsfeed`, all the newsfeed related code can be found in package `cz.pkopac.thesis.testing.dao.performance`[3]

```java
public interface INewsfeed {
  public Object fetchFollowedNewsFeed(long userId, int skip, int
    limit);
  public Object fetchGlobalNewsFeed(long userId, int skip, int
    limit);
}
```

[3]As you see the methods return only `Object` which would be too general for a real application, but there we want to know only how fast the requests are finished, we are not really interested in the data itself. Also, because the test was supposed to be automated, there was no need to visualize it, I tested that the data are correct only in the IDE.

Database Operations

To be able to switch Neo4j into batch mode on runtime and clean it, factory bean has been used, the `GraphDBManager`. This class is a singleton, which serves instances of the Neo4j database or `BatchInserter` according to the current mode set. Firstly, I also considered using the proxy pattern, but that would be unnecessarily complicated as the database API is really large. The important methods of this class are the following:

```java
public void setMode(boolean batch);

public GraphDatabaseService getDB();
public Neo4jTemplate getTemplate();
public BatchInserter getBatchInserter();

public void cleanStorage(String db_folder);
public void shutdownAll();
```

According to the mode set by the first method the other three always check, what instance of database to use with an internal method `checkBatchMode`. If the mode has changed, then the internally stored instances are shutdown and new ones are created and returned.

The second important class is `DBOperationsDAO`, which contains general operations over DBs. Its interface is the following:

```java
public interface IDBOperationsService {
  public void dropAndCreateAllTables();

  public void clearDuplicitiesInNeo4j(Relationships...
    relationships);

  public void restoreSQLIndexesAndRemoveDuplicates();

  public void clearMySQLCache();
}
```

6.3 Generator

The program is written in such a way that different generators could be used for testing using the DAOs and the user input with parameters of simulation contained in an instance of `Parameters`. The class is a normal bean, similar to DO classes, the only difference is that it has defined default values.

The interface for the generator is therefore very simple:

```java
public interface IGenerator {
  /* Definitions of constants for profiling */
  public void generate(Parameters p, Experiment e, AggStopWatch sw
    );
}
```

The first argument was already explained, the second one contains the instance of the experiment process, so that updates can be sent to it using the public method `setProgress(...)`. The last is the aggregating stop watch, which is used for measuring and storing profiling data.

In the implementation, called `GeneratorSimple`, the code is mostly procedural and the algorithms will be described in section 7.2. The part interesting from OOP point of view is maybe the generalized tree structure generator. As there are two hierarchical structures — the items and the groups, in the design process it was beneficial to use just one code to generate them both. The problem how to supply new instances for the algorithm was solved by using the factory pattern:

```java
public interface IFactory<T> {
  public T getNew(long id, long parentId, Parameters p);
}
```

An instance of the class implementing the proper factory is then passed to the algorithm. That effectively circumvents the problematic Java constructor.[4]

[4]In less strict languages, like JavaScript/ECMAScript it is possible to pass the constructor (or any function) as an argument.

6.4 Experiment

The `Experiment` class extends the Java `Thread` and contains the logic for executing one experimental run. It sets up the databases, runs the generator, runs the performance tests and saves the results in the bean `Results`, which is then transported to the user through the web interface. It is also responsible for any exception handling (default behavior is saving the exception into the `Results` object).

The logic responsible for the generation of random content is separated into the utility class `RCUtil`, which in turn uses a small utility Lorem Ipsum[5] and standard Java random number generation.

The results of the experiments are being held in the memory even after the experiment has ended, but all the references to other used objects are thrown away, so they do not stay in memory as well and the garbage collector can delete them. For the storage a small DAO is used, which internally uses just a `List` with the experiments and also returns the currently running experiment, so that two of them are not running at the same time.

[5]Lorem Ipsum Library can be found at: http://loremipsum.sourceforge.net/

7 Simulations

In this chapter we will shortly describe the course of one experiment, with interesting aspects of the implementation not directly related to the databases.

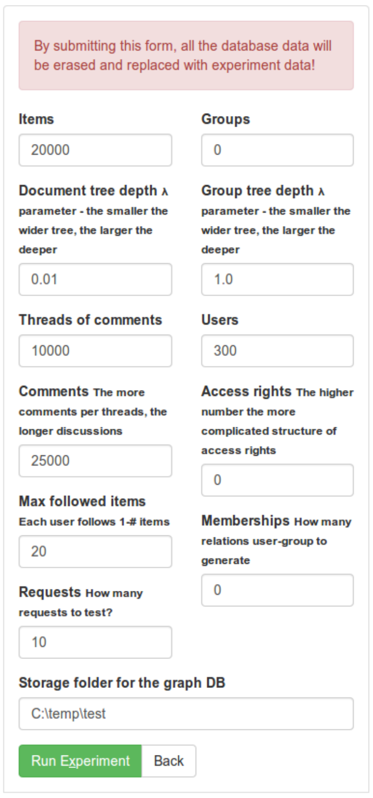
7.1 User Interface

To be user friendly the application offers a web interface for the user to put in the simulation parameters. This interface is very straightforward and explains the fields.

The technology behind this is the JSP with Spring bean-enhanced forms, where a domain object `Parameters` is bound to each field, is validated and then sent further inside the application, if the form is successfully submitted.

The handling is done by the Java class `ExperimentController`, which also handles requests for progress, because after the parameters are submitted, the user is redirected to a page, which loads the results dynamically from the server. As long as the results are not ready, the interface checks the current progress with the server and shows a progress bar.

The information about progress, including the results are fetched using *jQuery* call `$.getJSON()` from the server. On the server the mapping recognizes that the client requested a response of type *application/json*, so it returns the object `Results`. Thanks to the *Jackson* library and its extension *Databind* the object is automatically transformed into JSON and sent properly to the client.



By submitting this form, all the database data will be erased and replaced with experiment data!

Items 20000	**Groups** 0
Document tree depth λ parameter - the smaller the wider tree, the larger the deeper: 0.01	**Group tree depth λ** parameter - the smaller the wider tree, the larger the deeper: 1.0
Threads of comments 10000	**Users** 300
Comments The more comments per threads, the longer discussions: 25000	**Access rights** The higher number the more complicated structure of access rights: 0
Max followed items Each user follows 1-# items: 20	**Memberships** How many relations user-group to generate: 0
Requests How many requests to test?: 10	

Storage folder for the graph DB C:\temp\test

Figure 7.1: User Interface.

7.2 Data Generator

Once the experiment workflow is started by the user submitting the parameters and after the database connections are ready, the generator creates a dataset to use for later benchmarking of the databases.

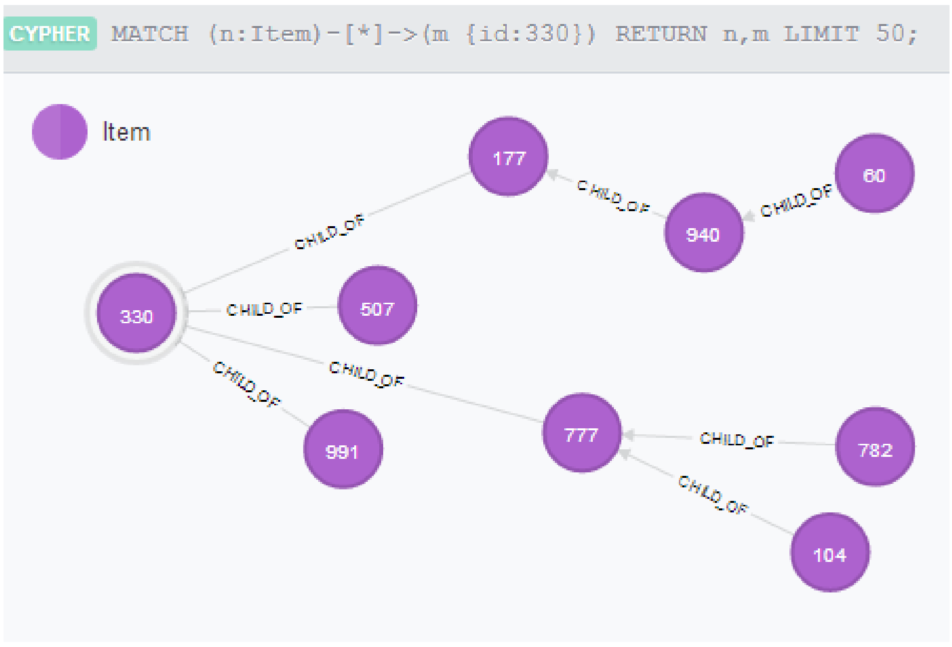
The mainly researched in the generator was the tree structure of the items and groups. As was previously mentioned, both the structures are generated by the same recursive algorithm. As the original code is too long, we can maybe have a look on a pseudocode only:

```
buildSubTree(parentId, available, subtreeDepth) {
  while (available > 0) {
    newId = generateId();
    createNew(newId, parentId);
    depth = randomDepth();
    if(depth > 0) { // Should a subtree be created?
      generate = randomBetween(1,available);
      buildSubTree(newId, generate, depth - 1)
      available -= generate;
    }
  }
}
```

This way we are generating items whose parents already exist[1] and we can influence very easily the average depth of the tree. Exponential probability distribution was used for the depth parameter, so that with the lower λ like 0.001 the tree is more wide and with closer to 1 it is more deep. Also the subtree sizes are random, with uniform distribution; making approximately in the beginning larger subtrees and smaller ones, when there are less elements available.

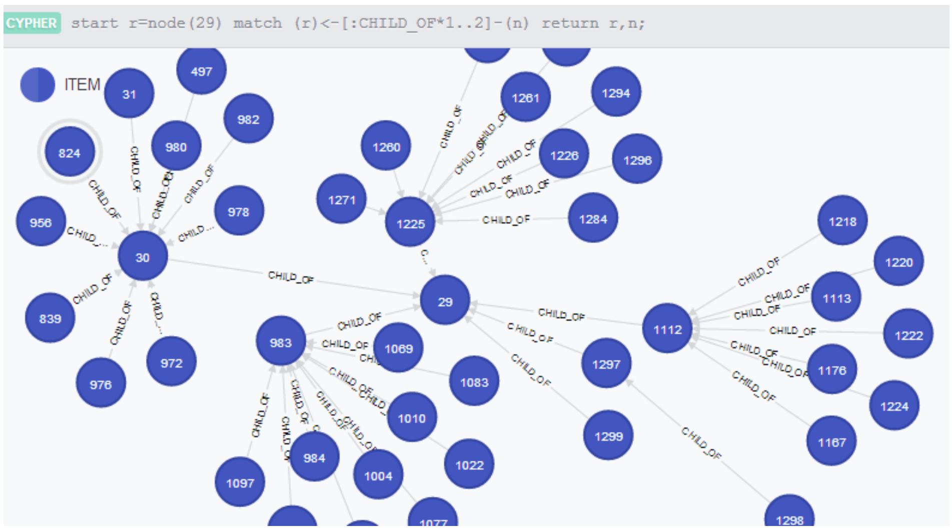
At first I used a very simple algorithm, which just randomly set the `parentId` in the valid range. But it was not possible to use for not fulfilling the condition of using only already existing parents and because it generated sparse and deep trees — all the nodes had the same uniform probability of having a child, so they had one or two. In such a system still there is one root, but it is very unnatural. The real item trees are more wider, with some subtrees bigger and some smaller. Comparison is on the next page.

[1]Important for saving relations into Neo4j, as the generator does not remember anything and is streamlined as much as possible.



```
MATCH (n:Item)-[*]->(m {id:330}) RETURN n,m LIMIT 50;
```

Figure 7.2: Naive document structure — as you can see this does not correspond with reality, where users tend to create lists, rather then tens or hundreds deep folders.



```
start r=node(29) match (r)<-[:CHILD_OF*1..2]-(n) return r,n;
```

Figure 7.3: Better generator used — the final generator creates a believable structure with some large subtrees with many lists of items and is also parameterizable.

Also, while generating random relations, one has to be aware of duplicities. Those should not be present, but can happen, as the application does not remember, what it has already generated (that would be too much data in memory). Especially with access rights and following.

For the SQL database it was solved by making the primary key out of the foreign keys in many-to-many tables (see section 5), but that would be not enough in itself. While uploading the data with `LOAD DATA INFILE` (section 5.4) the optional parameter `IGNORE` has to be used, so that duplicate entries are skipped.

With Neo4j and `BatchInserter` a different approach was taken — after everything is inserted, the following query was run, deleting all the duplicate relationships:

```
MATCH (c)-[r:REL_LABEL]->(t)
  WITH COLLECT(r) AS relIds, id(c) AS fir, id(t) AS sec, count(*)
    AS relCount
  WHERE relCount>1
  WITH tail(relIds) AS toDelIds,relIds,relCount
  FOREACH (d in toDelIds | DELETE d);
```

7.3 Gathering the Results

The user is presented at the end with time profiling information from the run. This is assembled by the class `AggStopWatch`, which was programmed to mimic the Spring `StopWatch` testing utility, which did offer simple counting of time, but had two problems — firstly, did not allow to count more tasks in parallel, secondly did not allow to run the same task again and again, adding the parts in a total sum. And this was what was needed, because some parts of the application are parallel.

When all the simulations were run a spreadsheet processor was used simply to make the final graph, even though originally it was planned to also make graphed results directly in the application. But unfortunately that was only an optional feature and much more time was spent on optimization of work with the databases.

Part III

Results and Conclusion

8 Methodology

While running the tests, I tried to make the conditions for the databases as close as possible, so that the results are comparable. It is important to say, however, that the differences between the databases can be influenced by many factors, above all the optimization of the queries and memory settings.

Most often the database performance on large data sets drops significantly, if there is not enough memory and the database has to access the disk, or even worse, pages of its memory get swapped to disk by the system. For both databases I allowed 5 GB of cache memory, which should be enough to hold tens of millions of records cached, it proved to be far more than needed, but in the server environment where the real application would run, using larger memory cache is not a big problem.

As Neo4j has separate cache settings I chose to put 4 GB to relationship store and 1 GB to node store — relationships are larger in memory and are necessary for fast traversals. Because Neo4j was running withing the same JVM as the application I also set higher heap size (8 GB).[1]

The cornerstone of the performance should be 25 000 items, as we have statistics available from the real application for several organizations with the largest having 18 000 items to date (see fig. 10.1).

The database performance tests were run with the same data sets each time generated randomly. After consultation with my supervisor we decided to run the worst case scenario tests, where there are no access rights set on the tree and therefore the algorithm must traverse to the top of the tree each time.

The application measured the time that it took to serve 10 requests made by random users (each time the same for both databases). Measuring used memory proved to be difficult, however; Neo4j was running with the application in the same JVM, so that it was not possible to distinguish, which part of the memory belongs to it, therefore memory complexity comparison is not included in the results.

[1]I was contacted by Michael Hunger, who works on Neo4j and he helped me with memory settings based on data analysis: he recommended to see how much the data takes on the hard drive and optimize accordingly to much lower values around hundreds of mega bytes to lower possible swapping. He was able to test the queries and reached much better results with the "followed" variant, but unfortunately I was not able to reproduce the same results.

9 Results

After all the tests were run, they were assembled into a chart, which you can see in fig. 9.1. The description is obvious enough to understand that for low numbers of items (under 25 000) both databases are basically performing well enough for a user not to notice any special waiting. We can even see that the times are comparable with the real application (see fig. 10.1), which is a good sign that the simulation is close to the original.

Unfortunately, the algorithm bounds are exponential and the performance graph shows that very well; with the growing number of items both databases are taking exponentially longer time to process the requests, until they take too much — no user would wait more than 5 seconds for a newsfeed.

This is the worst case scenario (but also default behavior of the real application) and the requests would probably take less than this in average in real data, but the important aspect is the trend we can see. Such purely recursive algorithm clearly can not be applied on a larger scale, where we mean to store and traverse millions of items and not even database caching can help that.

The fact that MySQL outperforms Neo4j in this case results probably from several facts. Firstly, the algorithm for the newsfeed and access rights, although recursive and graph-like, does not use any traditional graph algorithms, which is the really strong side of graph database optimizations. Secondly, from the discussion with Michael Hunger emerged the fact that in the current version of Neo4j there is a bug with unbounded-length paths, which makes their complexity $O(n^2)$ and will be corrected in the upcoming versions. These paths (recursive traversal, ie. transitive closure) are part of the queries that were used. He pointed out as well that the problem may be in the platform or the memory settings.

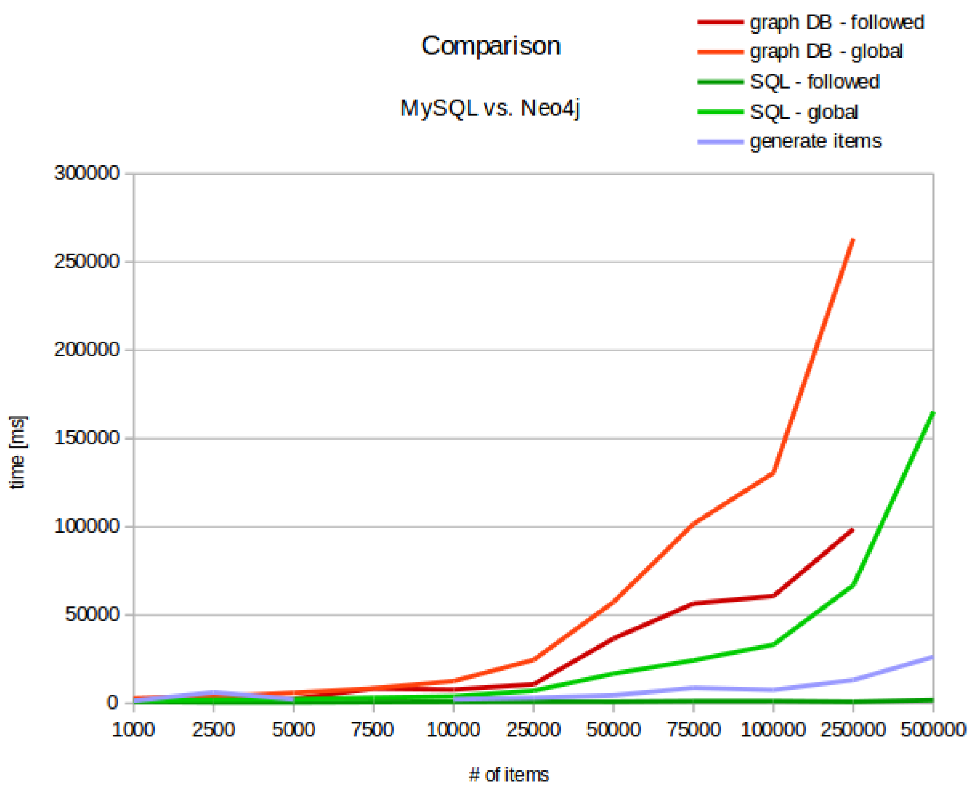


Figure 9.1: Results in a chart. On the axis y you can see the time that it took to serve 10 sequential requests. The times for serving the requests are growing exponentially for both the databases, which is expected, but Neo4j does grow faster than MySQL. Up to some 25 000 items the difference for the user is not significant, but at 250000 the graph database already takes approximately 25 seconds to fetch the global newsfeed. At the same magnitude MySQL can process the request in 7 seconds. Nevertheless, with 500 000 it does not make sense anymore to measure the graph database as well as with 750 000 items and MySQL. See appendix for the raw source data of the chart on page 83.

10 Conclusion

The experimental research and results in this thesis clearly show that should the application use the original algorithm with recursive check of access rights for each item, it will not perform sufficiently well for more than the current peak organization size. This project proved that the algorithm can not be optimized only by using a different database paradigm.

One of the possible solutions is to take away one aspect of the algorithm, which is the recursive access right check, because that is $O(Nlog(N))$ complexity, which are operations that can be spared, if we make a table caching the exact access rights on each node, using the recursive algorithm only once the rights change. In the graph database the equivalent would be to put specially labeled relationships.

This approach is actually under testing in Samepage right now and on the figure 10.1 you can see the significant drop in request processing times after the algorithm using the caching table was deployed. Of course, such caches potentially bring issues with data inconsistency — for example while changing access rights one would need to lock the cache table as well and run reconstruction for the affected subtree. But the users make many more requests on the newsfeed API than changing the access rights.

As the result of measurements and research in this thesis it was decided that Samepage will keep using SQL-based database and change the algorithm instead. After the testing of the new algorithm with direct access table is finished, it will be used in place of the current one.

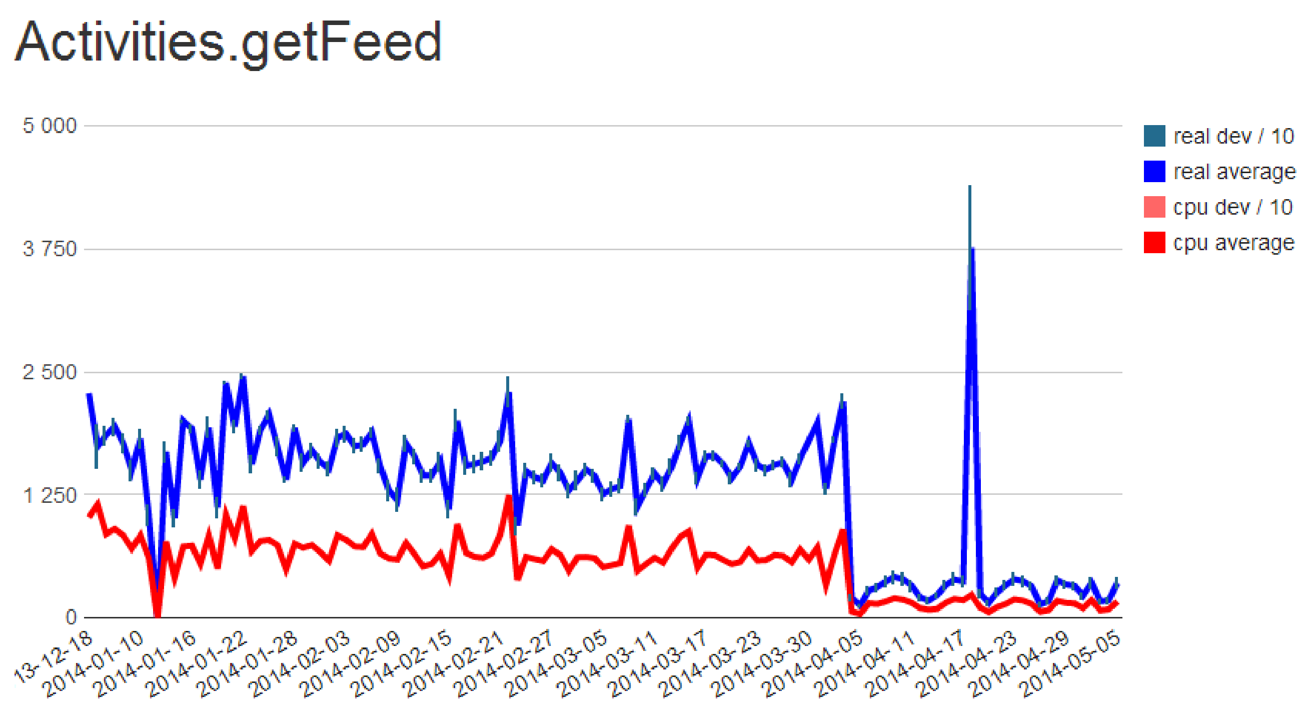


Figure 10.1: Statistics from Samepage.io application. On the y axis you can see the time in milliseconds that it took to serve one request in average. This data is taken from many organizations, most of which don't have many items. But there are also organizations, which are frequently used and have tens of thousands of items, being the most significant in the average. A clear drop in request serving time can be seen on April 5 2014, where the optimization was deployed, removing the recursive transitive closure to check access rights. (The very high peak on April 20 is a result of a fault and does not relate to this feature.)

Abbreviations

AOP Aspect-Oriented Programming. 30

API Application Programming Interface. 14, 16, 31, 32

BFS Breadth-First Search. 19, 54

BSON Binary JSON. 13

CLI Command Line Interface. 55

CSS Cascading Style Sheets. 28

CSV Comma-Separated Values. 46, 56, 59

CTE Common Table Expression. 12

DAO Data Access Object. 24, 28–30, 32, 58

DFS Depth-First Search. 19

DO Domain Object. 25

HTML Hyper Text Markup Language. 28

HTTP Hypertext Transfer Protocol. 16

IDE Integrated Development Environment. 23, 60

IoC Inversion of control. 23–25

JAR Java ARchive. 26

Java EE Java Enterprise Edition. 2

JAX-RS Java API for RESTful Web Services. 47

JDBC Java Database Connectivity. 11, 15, 25, 31, 50, 52

JS JavaScript. 28

JSON JavaScript Object Notation. 13, 64

JSP Java Server Pages. 28, 64

JVM Java Virtual Machine. 47, 69, 80

MVC Model-View-Controller. 21, 24

ODBC Open Database Connectivity. 15

OOP Object-Oriented Programming. 57

PL/SQL Procedural Language/Structured Query Language. 5, 10

RDBMS Relational Database Management Software. 10, 13, 21

REST REpresentational State Transfer. 16

SEQUEL Structured English Query Language. 10

SQL Structured Query Language. 4, 5, 8–10, 12, 21, 25, 29–33, 36

URL Uniform Resource Locator. 34

XML Extensible Markup Language. 24, 26

List of Figures

Bibliography

[Bichot(2013)] Guilhem Bichot. About mysql development: With recursive and mysql, 2013. URL `<http://guilhembichot.blogspot.co.uk/2013/11/with-recursive-and-mysql.html>`.

[Chamberlin(1974)] Donald D. Chamberlin. *SEQUEL: A Structured English Query Language*. Association for Computing Machinery, 1974.

[CTE(2014)] CTE. Hierarchical and recursive queries in sql: Common table expression, 2014. URL `<https://en.wikipedia.org/wiki/Common_table_expression#Common_table_expression>`.

[Hunger(2012a)] Michael Hunger. *Good Relationships - The Spring Data Neo4j Guide Book*. InfoQ, 2012a.

[Hunger(2012b)] Michael Hunger. Parallel batch inserter with neo4j imported 20 billion relationships on ec2, 2012b. URL `<http://jexp.de/blog/2012/10/parallel-batch-inserter-with-neo4j/>`.

[Hunger(2014)] Michael Hunger. Neo4j (csv) batch importer, 2014. URL `<https://github.com/jexp/batch-import/tree/20>`.

[IoC(2014)] IoC. Inversion of control, 2014. URL `<https://en.wikipedia.org/wiki/Inversion_of_control>`.

[Johnson(2014)] Rod Johnson. *Spring Framework Reference Documentation*. 4.0.3.release edition, 2014.

[Metsker(2006)] Steven John Metsker. *Design patterns in Java*. 2 edition, 2006. ISBN 0-321-33302-0.

[Neo4j(2014)] Neo4j. The neo4j manual, 2014. URL `<http://docs.neo4j.org/>`.

[Oracle(2014)] Oracle. Mysql 5.6 reference manual, 2014. URL `<http://dev.mysql.com/doc/refman/5.6/en/>`.

[Pecinovský(2007)] Rudolf Pecinovský. *Návrhové vzory*. 1 edition, 2007. ISBN 978-80-251-1582-4.

[Rolfe(2011)] Timothy Rolfe. Average node depth in a full binary tree, 2011. URL `<http://penguin.ewu.edu/cscd320/Topic/BSTree/BSTtwo/Full_Avg_Depth.pdf>`.

[Stevenson(2013)] Angus Stevenson, editor. *New Oxford American Dictionary*. Oxford University Press, 2013. ISBN 978-0199571123.

[Walls(2011)] Craig Walls. *Spring in action*. Manning Publications, 3rd edition, 2011. ISBN 978-1-935182-35-1.

A User Documentation

A.1 Setup of the Application

To run the application you need the following software installed on your work station in these or higher versions:

- Java 1.7
- Maven 3.0
- MySQL Server 5.6

The software is multiplatform and has been tested on Windows and Linux.[1]

The Neo4j database is downloaded and set up automatically, but you have to set up manually the MySQL database, adding a schema "pkopac" and a user "pkopac" with a password "password" (or you can change these values in the file src/main/resources/cz/pkopac/thesis/config/) and rebuild the application or change the respective file inside a built WAR file, if you already have one. This user has to have all privileges on that schema.

```
GRANT ALL PRIVILEGES ON 'pkopac'.* TO 'pkopac'@'localhost';
```

The following command will start embedded Tomcat server with the app.:

```
mvn jetty:run
```

Do not forget to set higher JVM heap memory, if you are planning to run the simulation with more then 100 000 nodes.

```
-Xmx3G -Xms3G -Xmn1G -XX:+UseConcMarkSweepGC -server
```

[1]On Windows it was not possible to clear the temporary database from the disk by deleting it due to the way that JVM on Windows handles the file locks, so each time you will have to use a distinct temporary folder and delete the files manually after you are finished.

A.2 Using

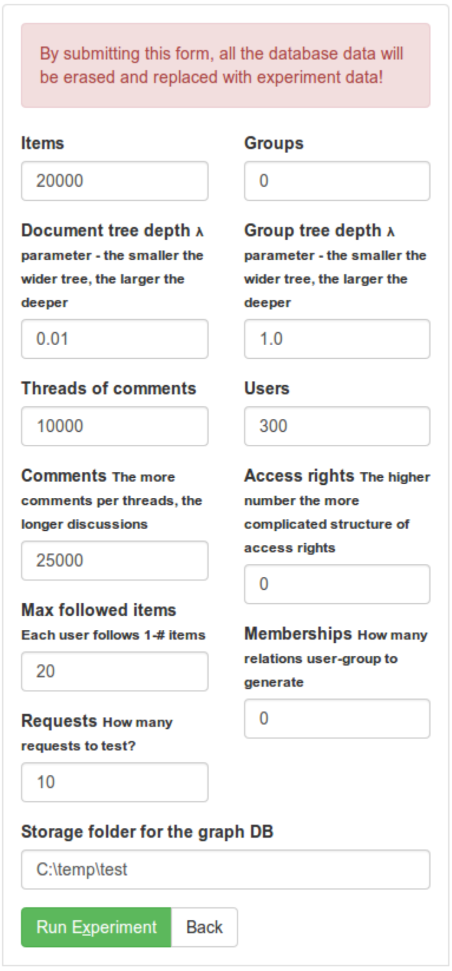
After you have successfully started the application with the command from the last section, you will see the following in the console:

```
[INFO] Started Jetty Server
```

Then, you can open your web browser and enter the address, where the application is listening:
http://localhost:8080/thesis

By clicking on the button (right upper corner in the menu) *Test Databases* –> *Run an experiment* you will reach the form displayed in the figure on the right. You can enter the following parameters:

- Number of items, groups, users, threads and replies — these are the ”activity” of the testing data.
- Tree depth for items and groups — try values like 1, 0.01, 0.001. The lower the value, the wider the tree and less deeper. (It is the λ parameter of exponential probability distribution.)
- Number of memberships, access rights and maximum followed items — these denote the complexity of the structure.
- How many requests to test.
- Where to save the temporary dataset of Neo4j (can be studied after the experiment is finished).


By submitting this form, all the database data will be erased and replaced with experiment data!

Items: 20000
Groups: 0
Document tree depth λ parameter - the smaller the wider tree, the larger the deeper: 0.01
Group tree depth λ parameter - the smaller the wider tree, the larger the deeper: 1.0
Threads of comments: 10000
Users: 300
Comments The more comments per threads, the longer discussions: 25000
Access rights The higher number the more complicated structure of access rights: 0
Max followed items Each user follows 1-# items: 20
Memberships How many relations user-group to generate: 0
Requests How many requests to test?: 10
Storage folder for the graph DB: C:\temp\test

Figure A.1: Input form with default values. Clicking on the button **Run Experiment** executes one experiment.

B Raw Result Data

Time [ms]	1000	2500	5000	7500	10000	25000	50000	75000	100000	250000	500000
Neo4j — F	2560	3638	2508	8265	7708	10751	36670	56518	60777	98698	
Neo4j — G	2792	3628	5932	8428	12577	24567	57336	101655	130561	263326	
MySQL — F	865	534	483	697	800	894	847	1164	1223	794	1823
MySQL — G	1466	2037	2336	3166	3869	7119	16778	24338	33211	66937	165249
generate items	1465	6169	2306	3541	2346	3092	4476	8751	7532	13196	26331